# MO401

IC/Unicamp

Prof Mario Côrtes

# Capítulo 4

# Data-Level Parallelism – Vector, SIMD, GPU

# Tópicos

- Vector architectures
- SIMD ISA extensions for multimedia
- GPU
- Detecting and enhancing loop level parallelism
- Crosscutting issues
- putting all together: mobile vs GPU, tesla....

# 4.1 Introduction

- SIMD architectures can exploit significant data-level parallelism for:
  - matrix-oriented scientific computing
  - media-oriented image and sound processors

- SIMD is more energy efficient than MIMD
  - Only needs to fetch one instruction per data operation
  - Makes SIMD attractive for personal mobile devices

- SIMD allows programmer to continue to think sequentially

# SIMD Parallelism

- Variations of SIMD
  - Vector architectures
    - Fácil de entender/programar; era considerado caro para microproc (área, DRAM bandwitdth)
  - SIMD extensions: multimedia → MMX, SSE, AVX
  - Graphics Processor Units (GPUs) → vector, many core heterog.

- For x86 processors:
  - Expect two additional cores per chip per year
  - SIMD width to double every four years
  - Potential speedup from SIMD to be twice that from MIMD!

# Speedup vs X86

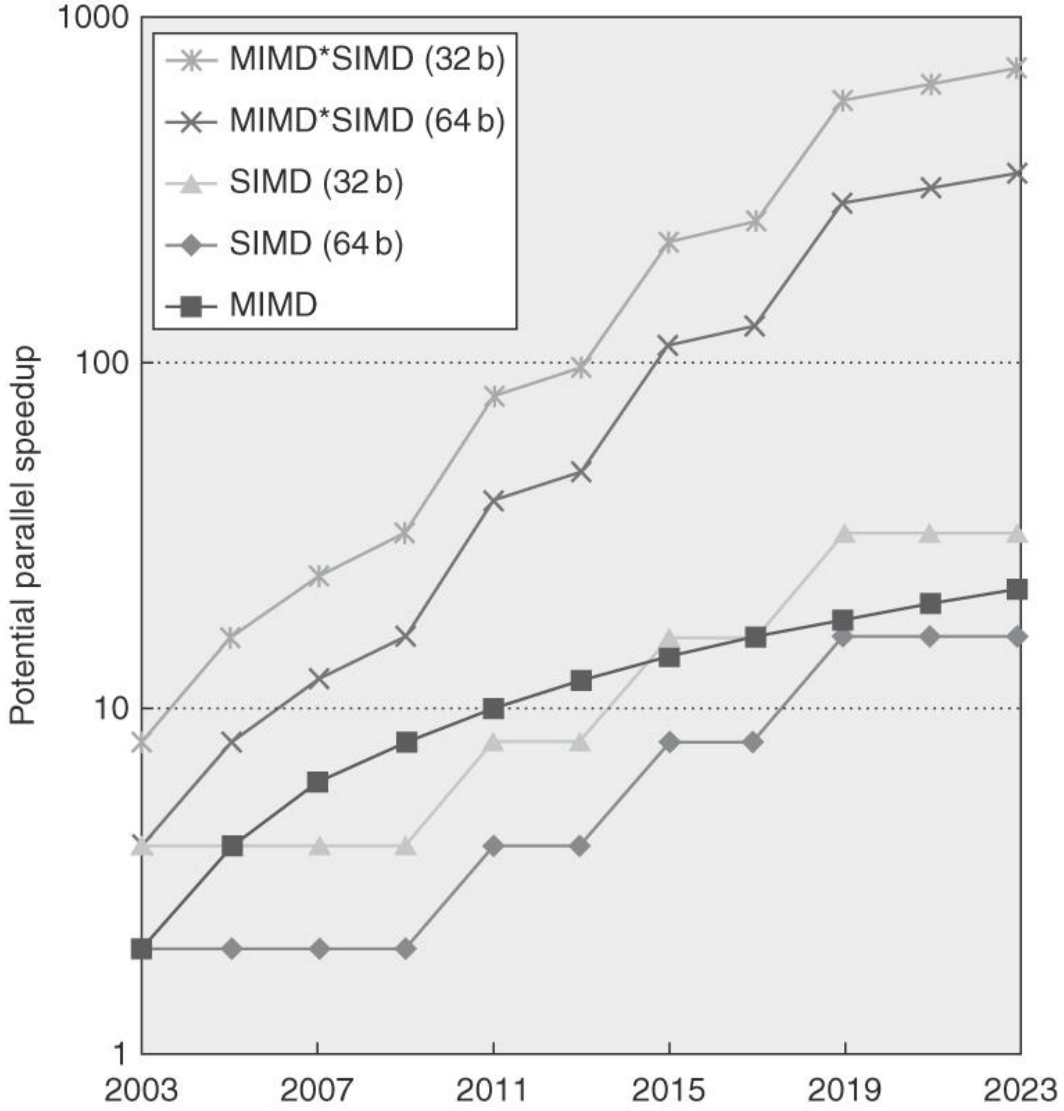


Figure 4.1 Potential speedup via parallelism from MIMD, SIMD, and both MIMD and SIMD over time for x86 computers. This figure assumes that two cores per chip for MIMD will be added every two years and the number of operations for SIMD will double every four years.

# 4.2 Vector Architectures

- Basic idea:
  - Read  (scattered) sets of data elements into “vector registers”
  - Operate on those registers
  - Disperse the results back into memory

- Registers are controlled by compiler
  - Used to hide memory latency
  - Leverage memory bandwidth
  - Loads e Stores → deeply pipelined
    - High latency, but high  hw utilization

# VMIPS

- Example architecture:  VMIPS
  - Loosely based on Cray-1
  - Vector registers (8)
    - Each register holds a 64-element, 64 bits/element vector
    - Register file has 16 read ports and 8 write ports
  - Vector functional units (5)
    - Fully pipelined
    - Data and control hazards are detected
  - Vector load-store unit
    - Fully pipelined
    - One word per clock cycle after initial latency
  - Scalar registers
    - 32 general-purpose registers
    - 32 floating-point registers

# VMIPS Archit.

For a 64 x 64b register file

- 64 x 64b elements
- 128 x 32b elements
- 256 x 16b elements
- 512 x 8b elements

Vector architecture is attractive both for scientific and multimedia apps

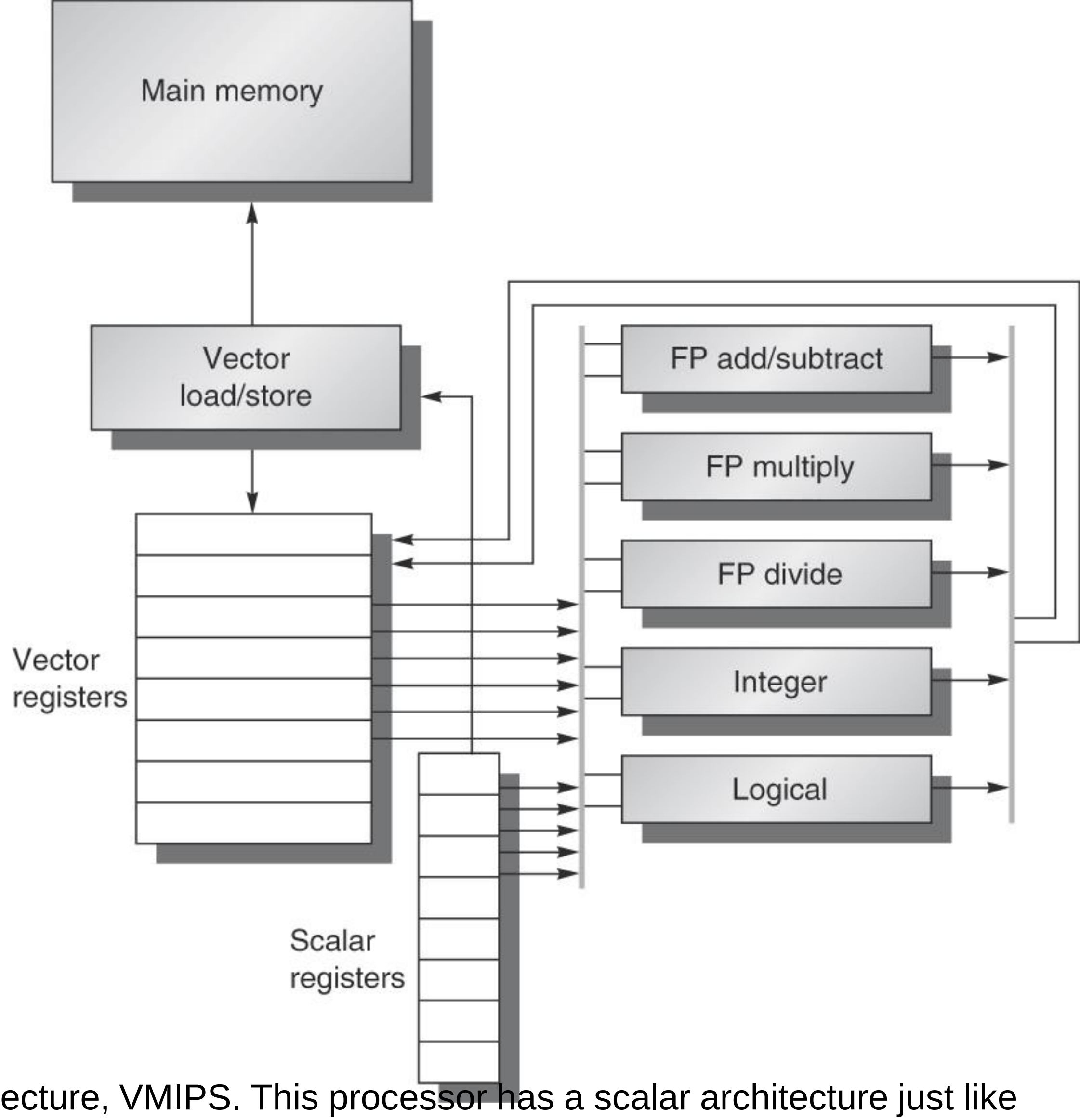
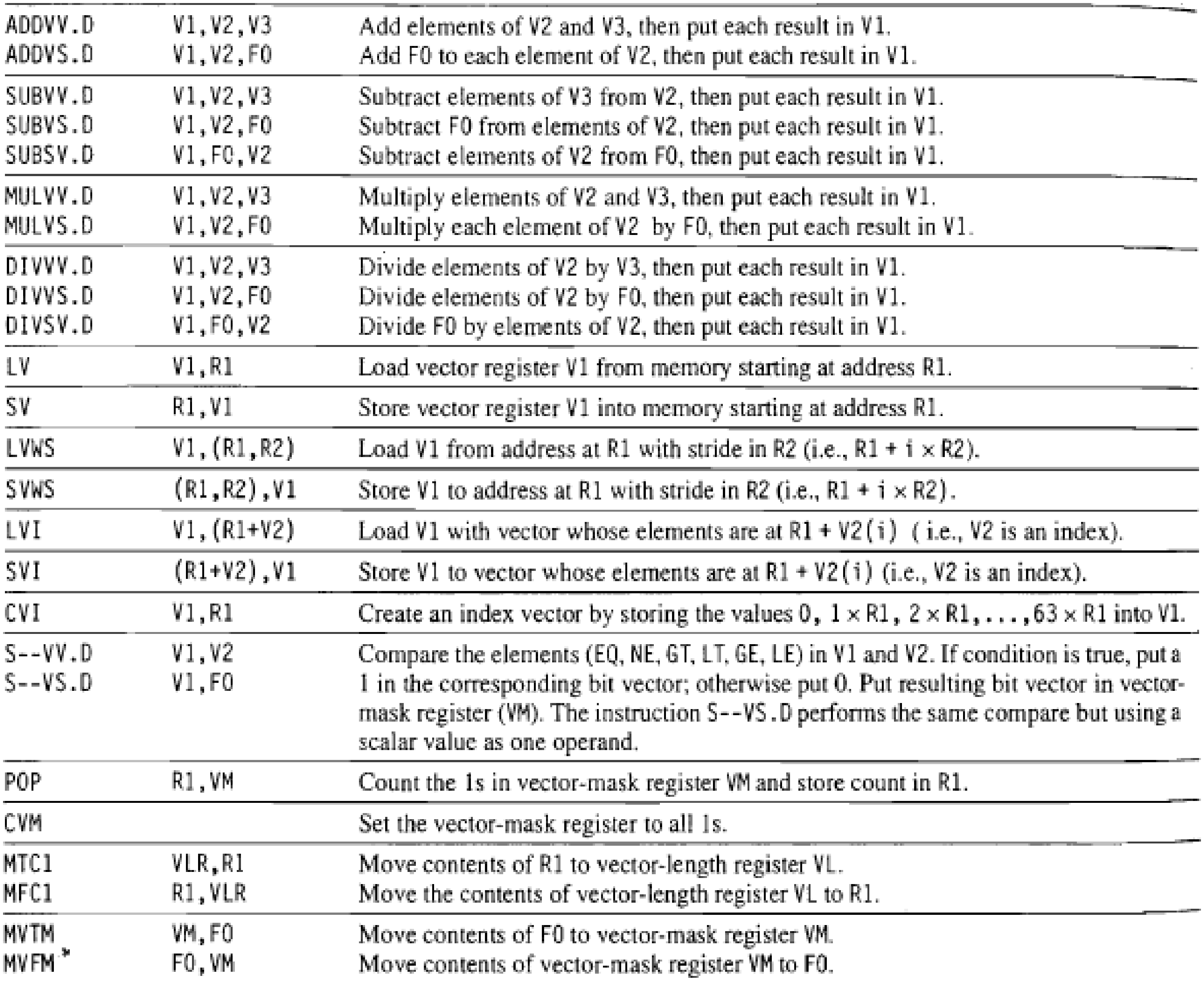


Figure 4.2 The basic structure of a vector architecture, VMIPS. This processor has a scalar architecture just like MIPS. There are also eight 64-element vector registers, and all the functional units are vector functional units. This chapter defines special vector instructions for both arithmetic and memory accesses. The figure shows vector units for logical and integer operations so that VMIPS looks like a standard vector processor that usually includes these units; however, we will not be discussing these units. The vector and scalar registers have a significant number of read and write ports to allow multiple simultaneous vector operations. A set of crossbar switches (thick gray lines) connects these ports to the inputs and outputs of the vector functional units.

# Fig 4.3 VMIPS ISA

VV:
vector – vector

VS:
vector – scalar



| | | |
|---|---|---|
| ADDVV.D | V1,V2,V3 | Add elements of V2 and V3, then put each result in V1. |
| ADDVS.D | V1,V2,F0 | Add F0 to each element of V2, then put each result in V1. |
| SUBVV.D | V1,V2,V3 | Subtract elements of V3 from V2, then put each result in V1. |
| SUBVS.D | V1,V2,F0 | Subtract F0 from elements of V2, then put each result in V1. |
| SUBSV.D | V1,F0,V2 | Subtract elements of V2 from F0, then put each result in V1. |
| MULVV.D | V1,V2,V3 | Multiply elements of V2 and V3, then put each result in V1. |
| MULVS.D | V1,V2,F0 | Multiply each element of V2 by F0, then put each result in V1. |
| DIVVV.D | V1,V2,V3 | Divide elements of V2 by V3, then put each result in V1. |
| DIVVS.D | V1,V2,F0 | Divide elements of V2 by F0, then put each result in V1. |
| DIVSV.D | V1,F0,V2 | Divide F0 by elements of V2, then put each result in V1. |
| LV | V1,R1 | Load vector register V1 from memory starting at address R1. |
| SV | R1,V1 | Store vector register V1 into memory starting at address R1. |
| LVWS | V1,(R1,R2) | Load V1 from address at R1 with stride in R2 (i.e., R1 + i × R2). |
| SVWS | (R1,R2),V1 | Store V1 to address at R1 with stride in R2 (i.e., R1 + i × R2). |
| LVI | V1,(R1+V2) | Load V1 with vector whose elements are at R1 + V2(i) ( i.e., V2 is an index). |
| SVI | (R1+V2),V1 | Store V1 to vector whose elements are at R1 + V2(i) (i.e., V2 is an index). |
| CVI | V1,R1 | Create an index vector by storing the values 0, 1 × R1, 2 × R1,...,63 × R1 into V1. |
| S--VV.D | V1,V2 | Compare the elements (EQ, NE, GT, LT, GE, LE) in V1 and V2. If condition is true, put a 1 in the corresponding bit vector; otherwise put 0. Put resulting bit vector in vector-mask register (VM). The instruction S--VS.D performs the same compare but using a scalar value as one operand. |
| S--VS.D | V1,F0 | |
| POP | R1,VM | Count the 1s in vector-mask register VM and store count in R1. |
| CVM | | Set the vector-mask register to all 1s. |
| MTC1 | VLR,R1 | Move contents of R1 to vector-length register VL. |
| MFC1 | R1,VLR | Move the contents of vector-length register VL to R1. |
| MVTM | VM,F0 | Move contents of F0 to vector-mask register VM. |
| MVFM* | F0,VM | Move contents of vector-mask register VM to F0. |

# Exmpl p267: VMIPS Instructions

Vector Architectures

- DAXPY: Double A x X Plus Y → AX+Y

```
L.D         F0,a        ; load scalar a
LV          V1,Rx       ; load vector X
MULVS.D     V2,V1,F0    ; vector-scalar multiply
LV          V3,Ry       ; load vector Y
ADDVV       V4,V2,V3    ; add
SV          Ry,V4       ; store the result
```

- VMIPS vs MIPS
  - Requires 6 instructions vs. almost 600 for MIPS (half is overhead)
  - RAW in MIPS: MUL.D → ADD.D → S.D
  - Stall in VMIPS: only for 1st vector element, then, smooth flow through pipeline

# Vector Execution Time

- Execution time depends on three factors:
  - Length of operand vectors
  - Structural hazards
  - Data dependencies
- VMIPS functional units consume one element per clock cycle
  - Execution time is approximately the vector length
- *Convoy*
  - Set of vector instructions that could potentially execute together
  - não devem conter hazard estrutural
- Tempo de execução proporcional ao # convoys

# Chaining and Chimes

Vector Architectures

- Sequences with read-after-write dependency hazards can be in the same convoy via *chaining*
- *Chaining*
  - Allows a vector operation to start as soon as the individual elements of its vector source operand become available (similar to forwarding)
- *Chime*
  - Unit of time to execute one convoy
  - *m* convoys executes in *m* chimes
  - For vector length of *n*, requires *m* x *n* clock cycles

# Exmpl p270: execution time

```
LV          V1,Rx           ;load vector X
MULVS.D     V2,V1,F0        ;vector-scalar multiply
LV          V3,Ry           ;load vector Y
ADDVV.D     V4,V2,V3        ;add two vectors
SV          Ry,V4           ;store the sum
```

Convoys:

| | | | |
|---|---|---|---|
| 1 | LV | MULVS.D | (V1 → chain ) |
| 2 | LV | ADDVV.D | (struct. haz. LV convoys 1, 2) |
| 3 | SV | | (struct. haz. LV convoys 2, 3) |

3 chimes, 2 FP ops per result, cycles per FLOP = 1.5

For 64 element vectors, requires 64 x 3 = 192 clock cycles

# Challenges

- Start up time
  - Pipeline latency of vector functional unit
  - Assume the same as Cray-1
    - Floating-point add => 6 clock cycles
    - Floating-point multiply => 7 clock cycles
    - Floating-point divide => 20 clock cycles
    - Vector load => 12 clock cycles
- Needed improvements:
  - > 1 element per clock cycle
  - Non-64 wide vectors
  - IF statements in vector code (conditional branches)
  - Memory system optimizations to support vector processors
  - Multiple dimensional matrices
  - Sparse matrices
  - Programming a vector computer

# Multiple Lanes: beyond 1 element / cycle

Element *n* of vector register *A* is “hardwired” to element *n* of vector register *B*

Allows for multiple hardware lanes

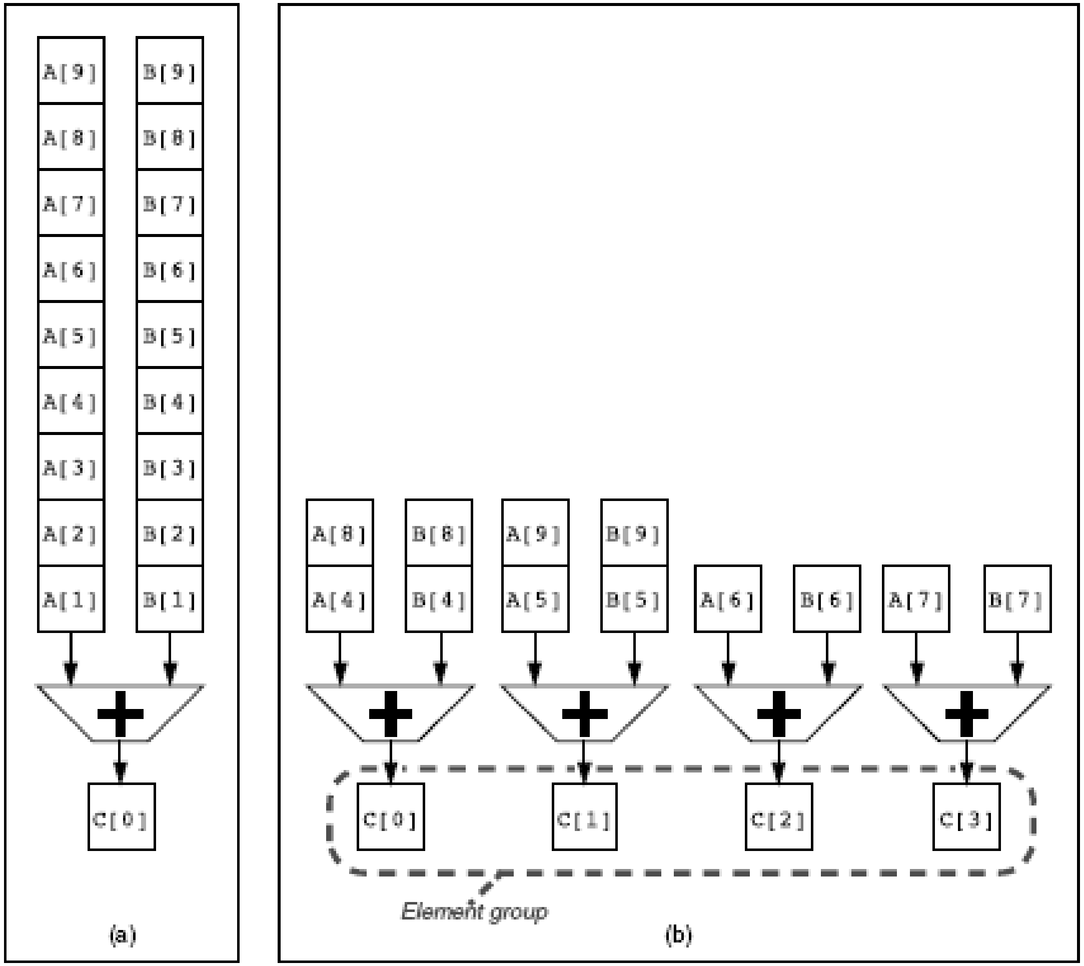


**Figure 4.4 Using multiple functional units to improve the performance of a single vector add instruction, C = A + B.** The vector processor (a) on the left has a single add pipeline and can complete one addition per cycle. The vector processor (b) on the right has four add pipelines and can complete four additions per cycle. The elements within a single vector add instruction are interleaved across the four pipelines. The set of elements that move through the pipelines together is termed an *element group*.

# Multiple Lanes: beyond 1 element / cycle

- 1 lane → 4 lanes
  - clocks in 1 chime: 64 → 16
- Multiple lanes:
  - little increase in complexity
  - no change in code
- Allows trade-off: area, clock rate, voltage, energy
  - ½ clock & 2x lanes→same speed

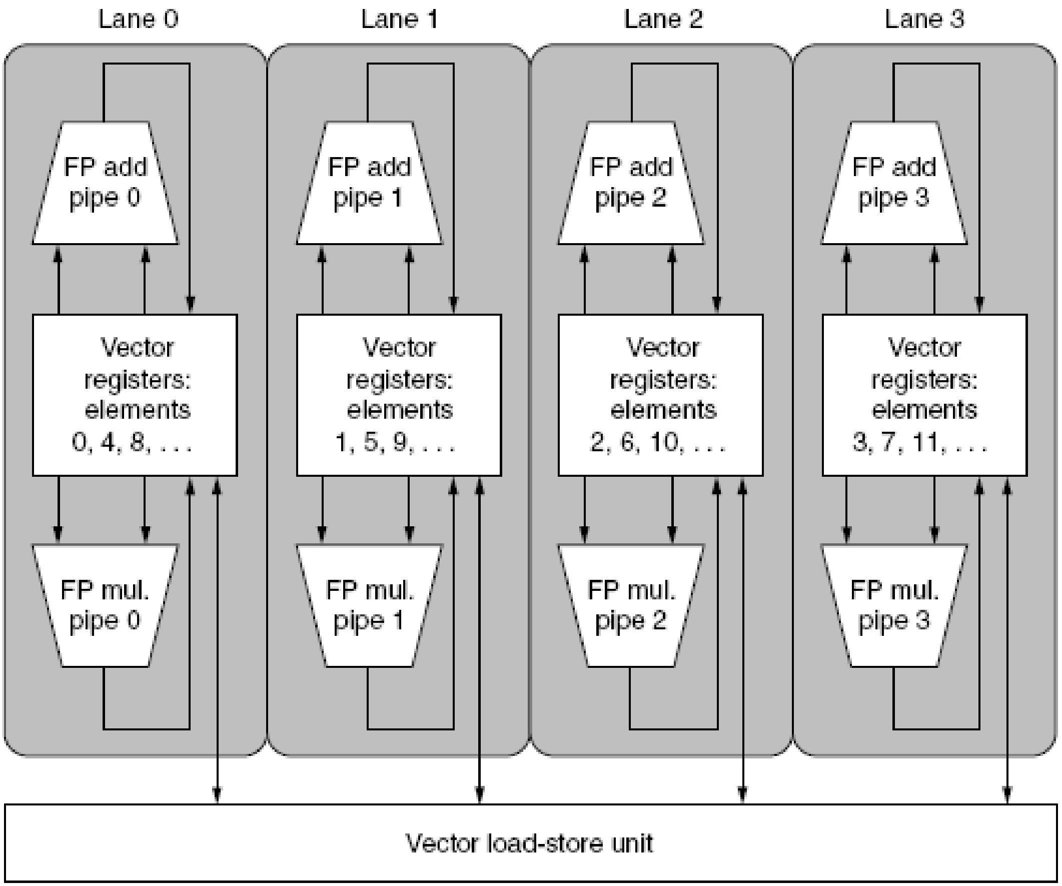
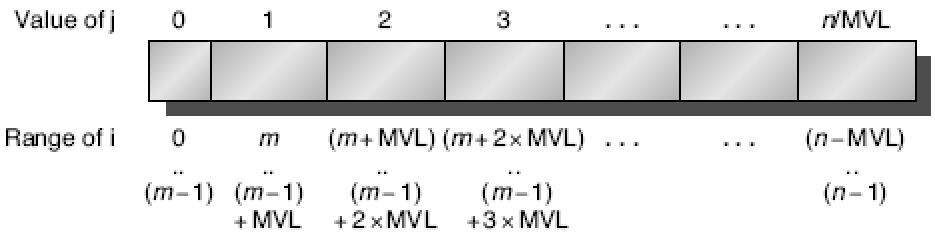

Lane 0
Lane 1
Lane 2
Lane 3
FP add pipe 0
FP add pipe 1
FP add pipe 2
FP add pipe 3
Vector registers: elements 0, 4, 8, . . .
Vector registers: elements 1, 5, 9, . . .
Vector registers: elements 2, 6, 10, . . .
Vector registers: elements 3, 7, 11, . . .
FP mul. pipe 0
FP mul. pipe 1
FP mul. pipe 2
FP mul. pipe 3
Vector load-store unit

**Figure 4.5 Structure of a vector unit containing four lanes.** The vector register storage is divided across the lanes, with each lane holding every fourth element of each vector register. The figure shows three vector functional units: an FP add, an FP multiply, and a load-store unit. Each of the vector arithmetic units contains four execution pipelines, one per lane, which act in concert to complete a single vector instruction. Note how each section of the vector register file only needs to provide enough ports for pipelines local to its lane. This figure does not show the path to provide the scalar operand for vector-scalar instructions, but the scalar processor (or control processor) broadcasts a scalar value to all lanes.

# Vector Length Register

- Vector length not known at compile time?
- Use Vector Length Register (VLR)
- O parâmetro MVL (max vector length) é usado pelo compilador →
  - não é necessário mudar ISA quando muda MVL (not in multimedia)
- Use strip mining for vectors over the maximum length:

```
low = 0;
VL = (n % MVL); /*find odd-size piece using modulo op % */
for (j = 0; j <= (n/MVL); j=j+1) { /*outer loop*/
      for (i = low; i < (low+VL); i=i+1) /*runs for length VL*/
         Y[i] = a * X[i] + Y[i] ; /*main operation*/
      low = low + VL; /*start of next vector*/
      VL = MVL; /*reset the length to maximum vector length*/
}
```



# Handling Ifs: Vector Mask Registers

- Consider:

```
for (i = 0; i < 64; i=i+1)
        if (X[i] != 0)
                X[i] = X[i] – Y[i];
```

- Use vector mask register to “disable” elements:

```
LV          V1,Rx          ;load vector X into V1
LV          V2,Ry          ;load vector Y
L.D         F0,#0          ;load FP zero into F0
SNEVS.D     V1,F0          ;sets VM(i) to 1 if V1(i)!=F0
SUBVV.D     V1,V1,V2       ;subtract under vector mask
SV          Rx,V1          ;store the result in X
```

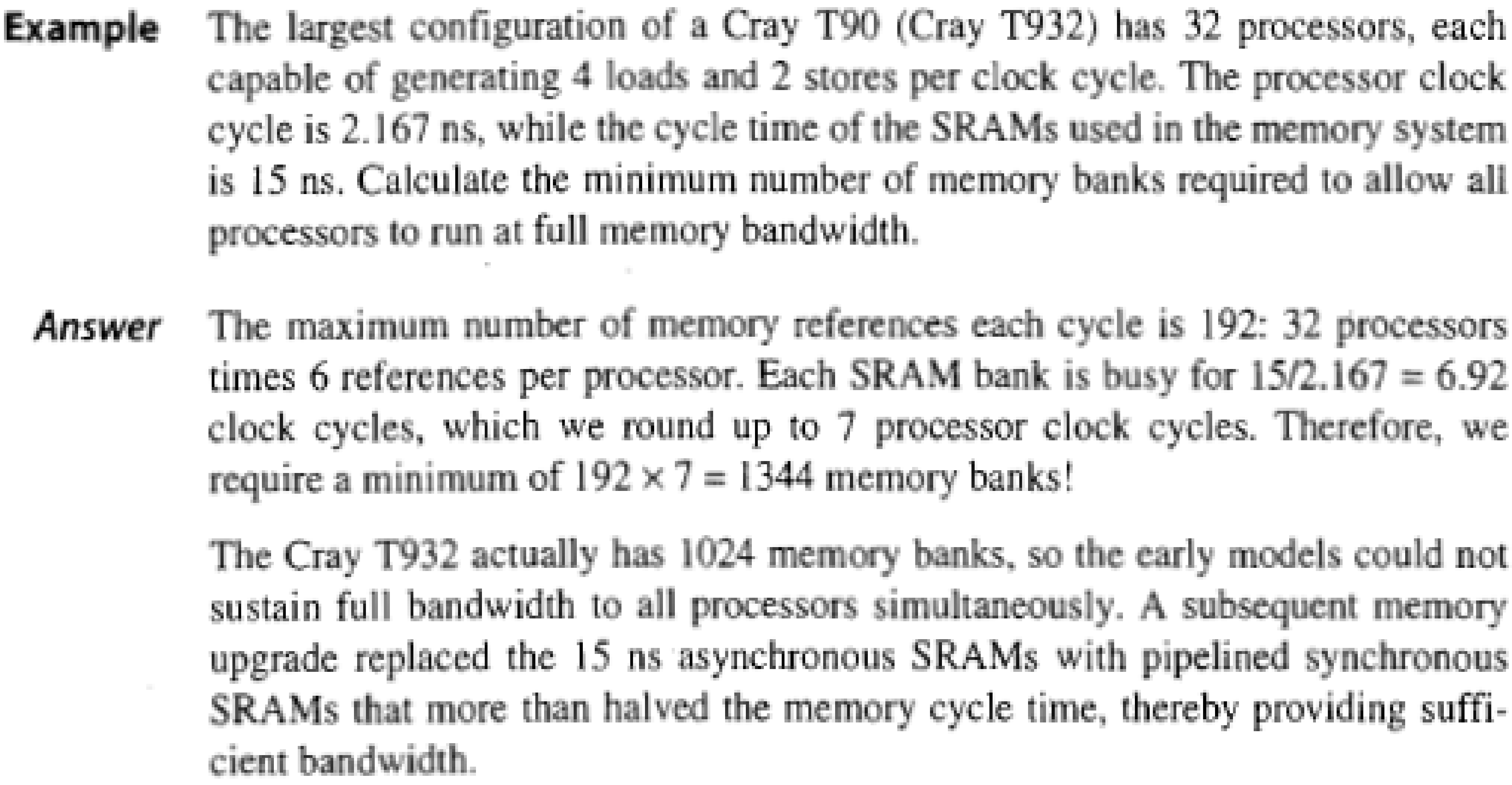
Set {NE} Vect x Scalar

- GFLOPS rate decreases!
  - additional instructions executed anyway (when vect mask reg is used)

# Memory Banks

- Memory system must be designed to support high bandwidth for vector loads and stores
- LD/ST vector unit: more complicated than arithmetic unit
  - Startup time: 1st word → register
    - typical penality: 100 cycles (12 cycles no VMIPS)
  - Initiation rate: reading rate from memory (could be ≠ 1 cycle)
    - for 1 / cycle: multiple memory banks
- Spread accesses across multiple banks
  - Control bank addresses independently
  - Ability to load or store non sequential words (not interleaving)
  - Support multiple vector processors sharing the same memory

# Exmpl p277: # of memory banks of Cray T90



**Example** The largest configuration of a Cray T90 (Cray T932) has 32 processors, each capable of generating 4 loads and 2 stores per clock cycle. The processor clock cycle is 2.167 ns, while the cycle time of the SRAMs used in the memory system is 15 ns. Calculate the minimum number of memory banks required to allow all processors to run at full memory bandwidth.

***Answer*** The maximum number of memory references each cycle is 192: 32 processors times 6 references per processor. Each SRAM bank is busy for 15/2.167 = 6.92 clock cycles, which we round up to 7 processor clock cycles. Therefore, we require a minimum of 192 × 7 = 1344 memory banks!

The Cray T932 actually has 1024 memory banks, so the early models could not sustain full bandwidth to all processors simultaneously. A subsequent memory upgrade replaced the 15 ns asynchronous SRAMs with pipelined synchronous SRAMs that more than halved the memory cycle time, thereby providing sufficient bandwidth.

# Stride: handling muldimentional arrays

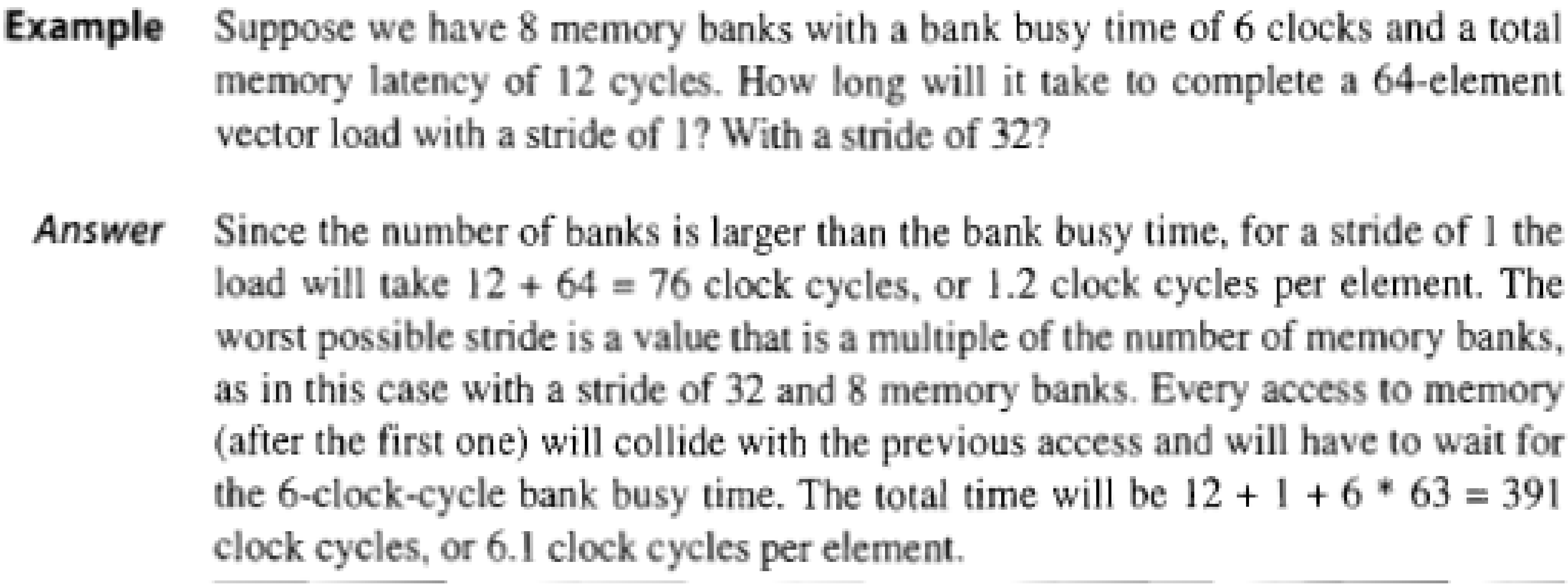
- Consider the matrix multiply: A = B * D

```
for (i = 0; i < 100; i=i+1)
    for (j = 0; j < 100; j=j+1) {
        A[i][j] = 0.0;
        for (k = 0; k < 100; k=k+1)
        A[i][j] = A[i][j] + B[i][k] * D[k][j];
    }
```

- 3D array stored as linear array in memory (row or column major)
  - one of B or D will have non adjacent elements in memory (row or column)
- Must vectorize multiplication of rows of B with columns of D
  - D elements separated by RowSize x EntrySize = 100 * 8 = 800 = stride
- Use non-unit stride → separated elements become contiguous in Vect Register (locality? better than cache?)
- Bank conflict (stall) if same bank is hit faster than bank busy time:

$$\frac{\text{N\_banks}}{\text{Min\_mult\_comum(Stride, N\_banks)}} < \text{Bank\_busy\_time}$$

# Exmpl p 279



**Example** Suppose we have 8 memory banks with a bank busy time of 6 clocks and a total memory latency of 12 cycles. How long will it take to complete a 64-element vector load with a stride of 1? With a stride of 32?

***Answer*** Since the number of banks is larger than the bank busy time, for a stride of 1 the load will take 12 + 64 = 76 clock cycles, or 1.2 clock cycles per element. The worst possible stride is a value that is a multiple of the number of memory banks, as in this case with a stride of 32 and 8 memory banks. Every access to memory (after the first one) will collide with the previous access and will have to wait for the 6-clock-cycle bank busy time. The total time will be 12 + 1 + 6 * 63 = 391 clock cycles, or 6.1 clock cycles per element.

# Gather-Scatter: Sparse Matrices

- Sparse vectors are usually stored in compacted form
- Consider:

```
for (i = 0; i < n; i=i+1)
      A[K[i]] = A[K[i]] + C[M[i]];
```

- Where K and M designate non-zero elements of A and C
  - K and M: same size
- Must be able to
  - gather: index vector allows loading to a dense vector
  - scatter: store back in memory in the expanded form (not compacted)
- HW support to Gather-Scatter: present in all modern vector processors. In VMIPS:
  - LVI (Load Vector Indexed – Gather)
  - SVI (Store Vector Indexed – Scatter)

# Gather-Scatter: Sparse Matrices (cont)

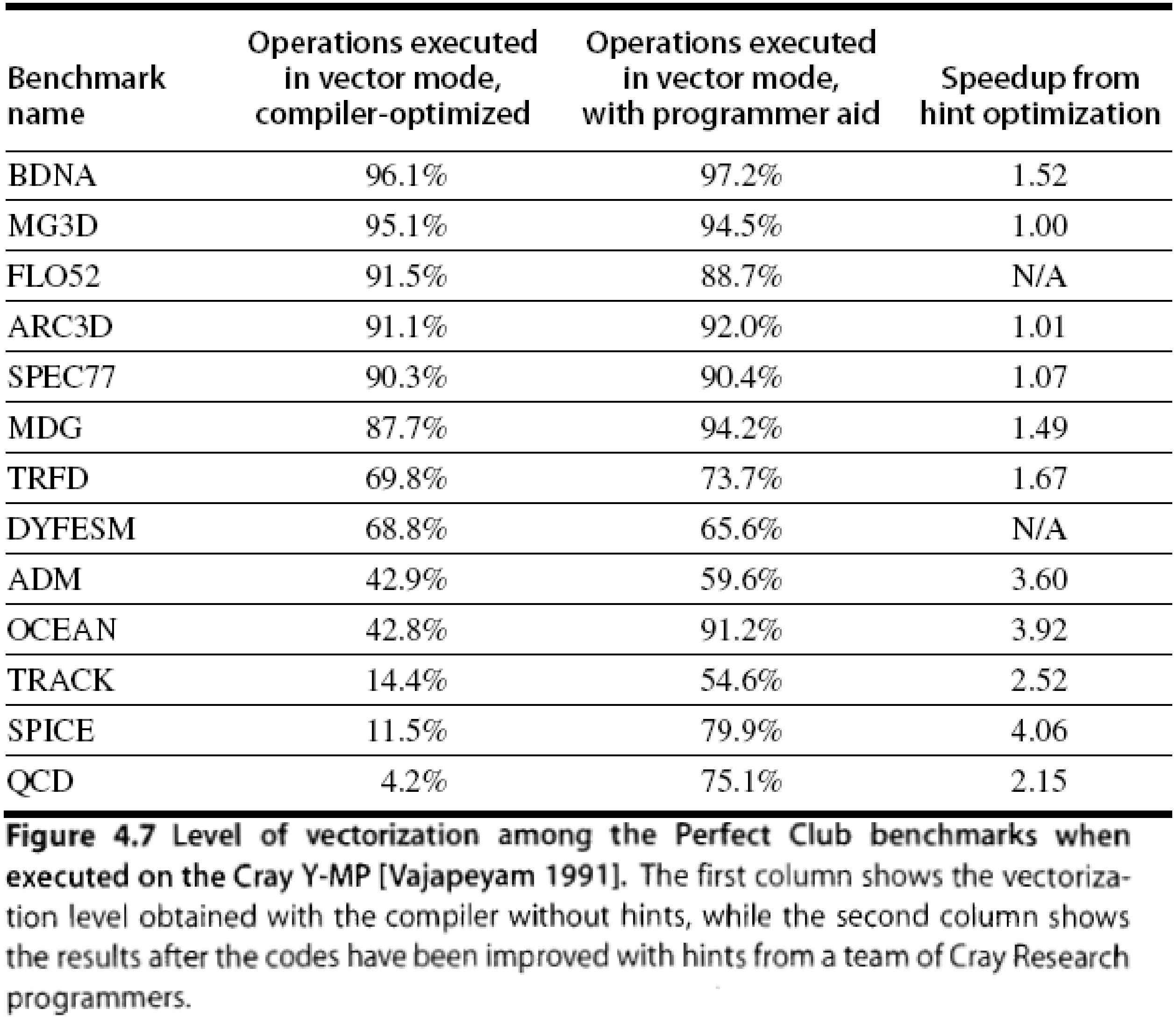
```
for (i = 0; i < n; i=i+1)
    A[K[i]] = A[K[i]] + C[M[i]];
```

- Ra, Rc, Rk, Rm:
  - starting vector addresses
- Use index vector:

```
LV       Vk, Rk          ;load K
LVI      Va, (Ra+Vk)     ;load A[K[]]
LV       Vm, Rm          ;load M
LVI      Vc, (Rc+Vm)     ;load C[M[]]
ADDVV.D  Va, Va, Vc      ;add them
SVI      (Ra+Vk), Va     ;store A[K[]]
```

# Programming Vec. Architectures

- Compilers can provide feedback to programmers
- Programmers can provide hints to compiler

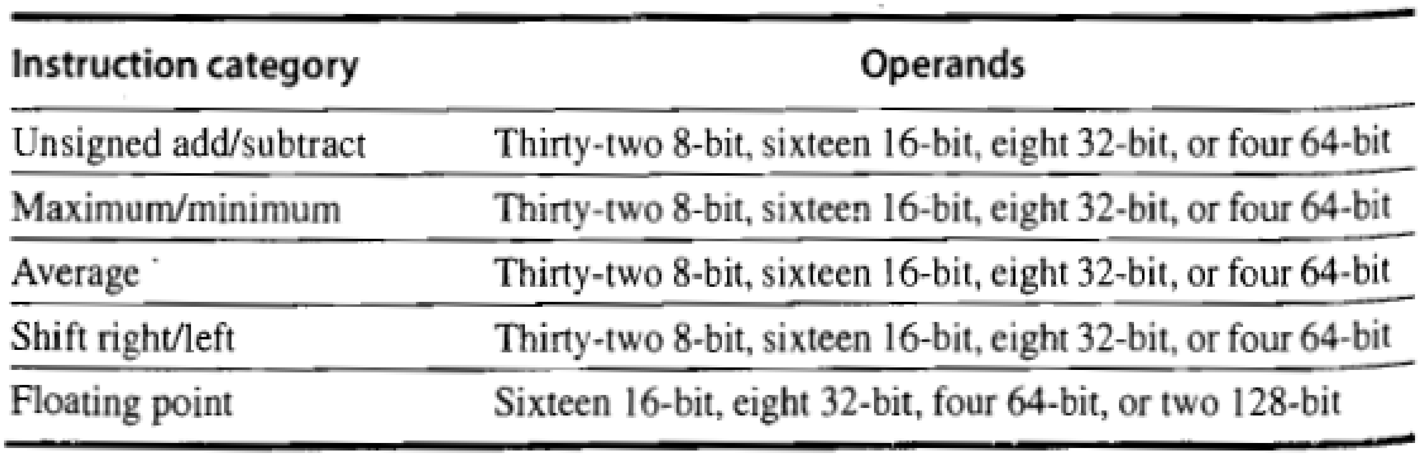


| Benchmark name | Operations executed in vector mode, compiler-optimized | Operations executed in vector mode, with programmer aid | Speedup from hint optimization |
|---|---|---|---|
| BDNA | 96.1% | 97.2% | 1.52 |
| MG3D | 95.1% | 94.5% | 1.00 |
| FLO52 | 91.5% | 88.7% | N/A |
| ARC3D | 91.1% | 92.0% | 1.01 |
| SPEC77 | 90.3% | 90.4% | 1.07 |
| MDG | 87.7% | 94.2% | 1.49 |
| TRFD | 69.8% | 73.7% | 1.67 |
| DYFESM | 68.8% | 65.6% | N/A |
| ADM | 42.9% | 59.6% | 3.60 |
| OCEAN | 42.8% | 91.2% | 3.92 |
| TRACK | 14.4% | 54.6% | 2.52 |
| SPICE | 11.5% | 79.9% | 4.06 |
| QCD | 4.2% | 75.1% | 2.15 |

**Figure 4.7 Level of vectorization among the Perfect Club benchmarks when executed on the Cray Y-MP [Vajapeyam 1991].** The first column shows the vectorization level obtained with the compiler without hints, while the second column shows the results after the codes have been improved with hints from a team of Cray Research programmers.

# 4.SIMD Extensions

- Media applications operate on data types narrower than the native word size
  - Many graphics systems: 8b (3 colors) + 8b (transparency)
  - Audio samples: 8 ou 16 bit
- Hardware changes (example)
  - Disconnect carry chains to “partition” adder: 8, 16, 32 bits



| Instruction category | Operands |
|---|---|
| Unsigned add/subtract | Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit |
| Maximum/minimum | Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit |
| Average | Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit |
| Shift right/left | Thirty-two 8-bit, sixteen 16-bit, eight 32-bit, or four 64-bit |
| Floating point | Sixteen 16-bit, eight 32-bit, four 64-bit, or two 128-bit |

**Figure 4.8 Summary of typical SIMD multimedia support for 256-bit-wide operations.** Note that the IEEE 754-2008 floating-point standard added half-precision (16-bit) and quad-precision (128-bit) floating-point operations.

SIMD Instruction Set Extensions for Multimedia

# Limitations of SIMD Extensions (vs Vector)

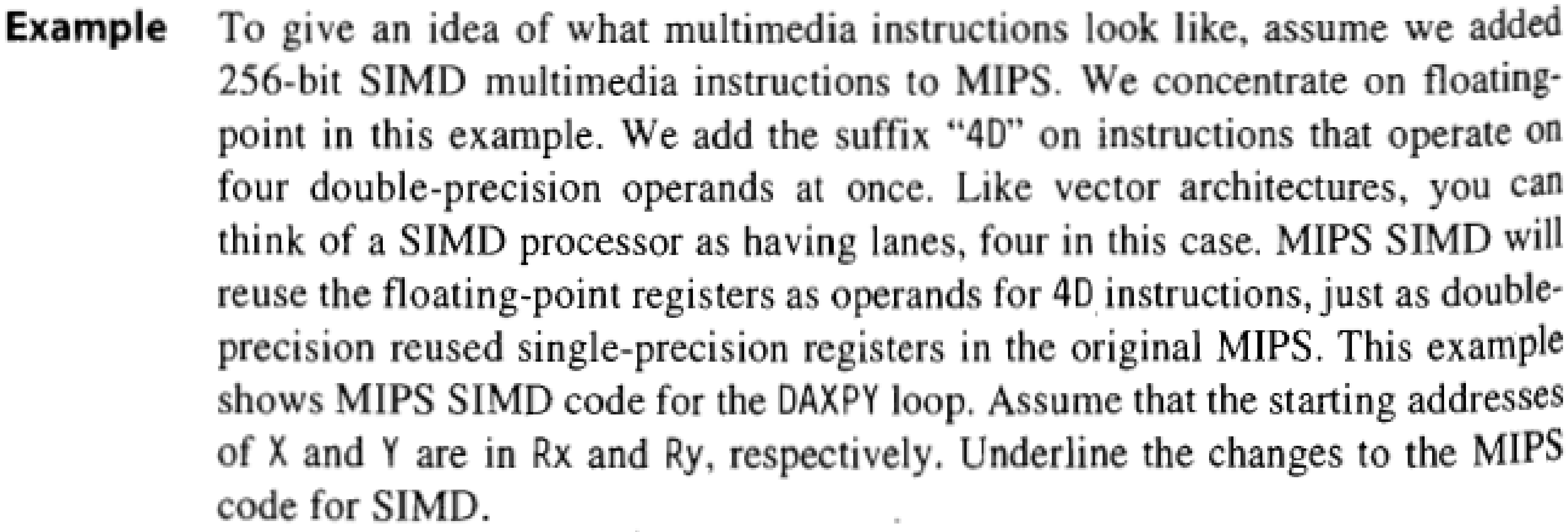
- Smaller register file
- Number of data operands encoded into op code (no Vector Length Register) → addition of 100´s of new op codes
- No sophisticated addressing modes (strided, scatter-gather)
  - fewer programs can be vectorized in SIMD extension machines
- No mask registers
- → increased difficulty of programming in SIMD assembly language

# SIMD Implementations

- Implementations:
  - Intel MMX (1996)
    - Eight 8-bit integer ops or four 16-bit integer ops
  - Streaming SIMD Extensions (SSE) (1999)
    - Eight 16-bit integer ops
    - Four 32-bit integer/fp ops or two 64-bit integer/fp ops
  - Advanced Vector Extensions (2010)
    - Four 64-bit integer/fp ops
- Goal: accelerate carefully written libraries (rather than for the compiler to generate them
- With so many flaws, why are SIMD so popular?
  - HW changes: easy, low cost, low area
  - No need of high memory BW (Vector)
  - Fewer problems with virtual memory and page faults (short vectors)

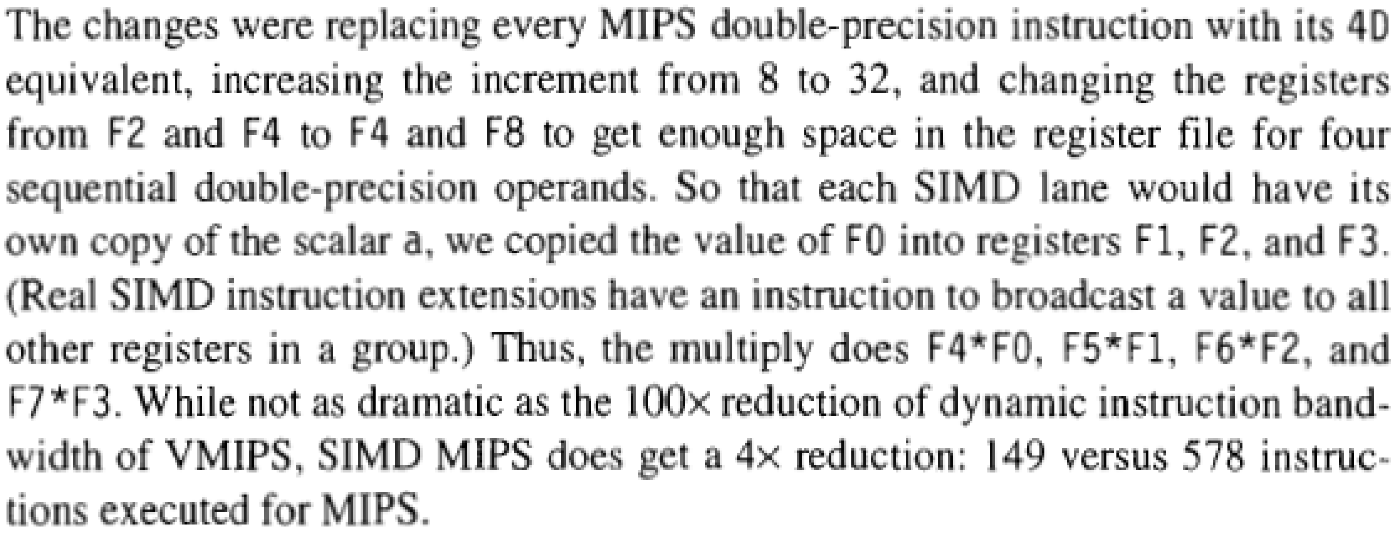
SIMD Instruction Set Extensions for Multimedia

# Exmpl p284: SIMD Code



**Example** To give an idea of what multimedia instructions look like, assume we added 256-bit SIMD multimedia instructions to MIPS. We concentrate on floating-point in this example. We add the suffix "4D" on instructions that operate on four double-precision operands at once. Like vector architectures, you can think of a SIMD processor as having lanes, four in this case. MIPS SIMD will reuse the floating-point registers as operands for 4D instructions, just as double-precision reused single-precision registers in the original MIPS. This example shows MIPS SIMD code for the DAXPY loop. Assume that the starting addresses of X and Y are in Rx and Ry, respectively. Underline the changes to the MIPS code for SIMD.

Anwser (next page)



The changes were replacing every MIPS double-precision instruction with its 4D equivalent, increasing the increment from 8 to 32, and changing the registers from F2 and F4 to F4 and F8 to get enough space in the register file for four sequential double-precision operands. So that each SIMD lane would have its own copy of the scalar a, we copied the value of F0 into registers F1, F2, and F3. (Real SIMD instruction extensions have an instruction to broadcast a value to all other registers in a group.) Thus, the multiply does F4*F0, F5*F1, F6*F2, and F7*F3. While not as dramatic as the 100× reduction of dynamic instruction bandwidth of VMIPS, SIMD MIPS does get a 4× reduction: 149 versus 578 instructions executed for MIPS.

# SIMD Code – DAXPY

```
      L.D      F0,a          ;load scalar a
      MOV      F1, F0        ;copy a into F1 for SIMD MUL
      MOV      F2, F0        ;copy a into F2 for SIMD MUL
      MOV      F3, F0        ;copy a into F3 for SIMD MUL
      DADDIU   R4,Rx,#512    ;last address to load
Loop: L.4D     F4,0[Rx]      ;load X[i], X[i+1], X[i+2], X[i+3]
      MUL.4D   F4,F4,F0      ;a×X[i],a×X[i+1],a×X[i+2],a×X[i+3]
      L.4D     F8,0[Ry]      ;load Y[i], Y[i+1], Y[i+2], Y[i+3]
      ADD.4D   F8,F8,F4      ;a×X[i]+Y[i], ..., a×X[i+3]+Y[i+3]
      S.4D     0[Ry],F8      ;store into Y[i], Y[i+1], Y[i+2], Y[i+3]
      DADDIU   Rx,Rx,#32     ;increment index to X
      DADDIU   Ry,Ry,#32     ;increment index to Y
      DSUBU    R20,R4,Rx     ;compute bound
      BNEZ     R20,Loop      ;check if done
```

# Roofline Performance Model

- Basic idea:
  - Plot peak floating-point throughput as a function of arithmetic intensity
  - Ties together floating-point performance and memory performance for a target machine
- Arithmetic intensity
  - Peak # Floating-point operations / Peak # data bytes transfered

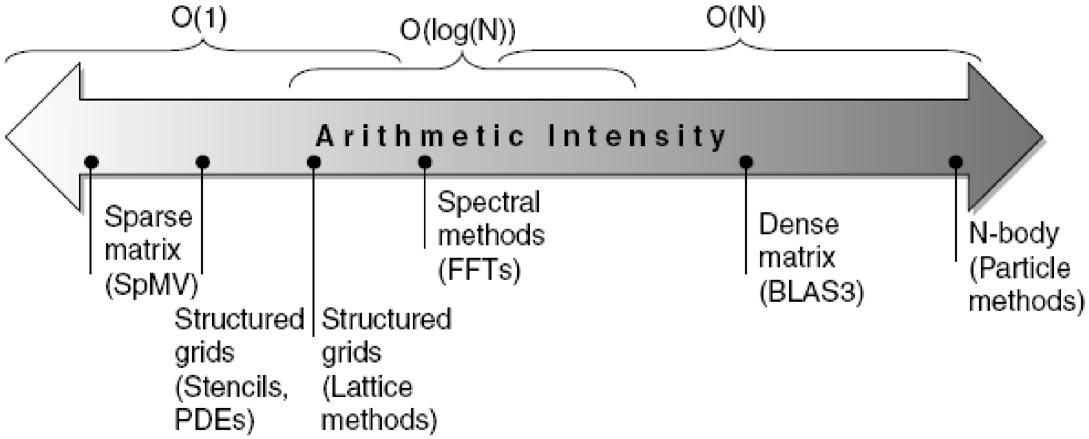


**Figure 4.10 Arithmetic intensity, specified as the number of floating-point operations to run the program divided by the number of bytes accessed in main memory [Williams et al. 2009].** Some kernels have an arithmetic intensity that scales with problem size, such as dense matrix, but there are many kernels with arithmetic intensities independent of problem size.

Cumieira
far left or far right?

# Examples

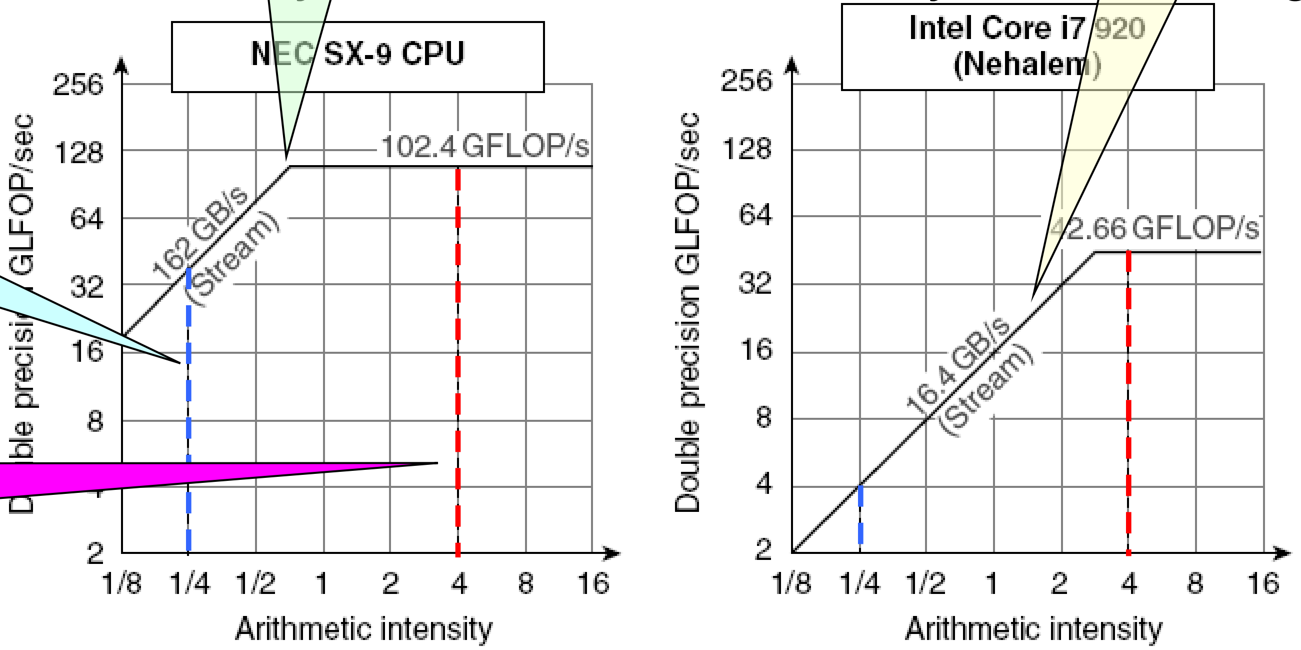
FLOP/s = B/sec
FLOP/B

- Attainable GFLOPs/sec =
  - Min(Peak Memory BW × Arithmetic Intensity, Peak Floating Point Perf.)

GFLOP/s

Memory bound

CPU bound



FLOP/Byte

**Figure 4.11 Roofline model for one NEC SX-9 vector processor on the left and the Intel Core i7 920 multicore computer with SIMD Extensions on the right [Williams et al. 2009].** This Roofline is for unit-stride memory accesses and double-precision floating-point performance. NEC SX-9 is a vector supercomputer announced in 2008 that costs millions of dollars. It has a peak DP FP performance of 102.4 GFLOP/sec and a peak memory bandwidth of 162 GBytes/sec from the Stream benchmark. The Core i7 920 has a peak DP FP performance of 42.66 GFLOP/sec and a peak memory bandwidth of 16.4 GBytes/sec. The dashed vertical lines at an arithmetic intensity of 4 FLOP/byte show that both processors operate at peak performance. In this case, the SX-9 at 102.4 FLOP/sec is 2.4x faster than the Core i7 at 42.66 GFLOP/sec. At an arithmetic intensity of 0.25 FLOP/byte, the SX-9 is 10x faster at 40.5 GFLOP/sec versus 4.1 GFLOP/sec for the Core i7.

# 4.4 Graphical Processing Units

- Given the hardware invested to do graphics well, how can be supplement it to improve performance of a wider range of applications?

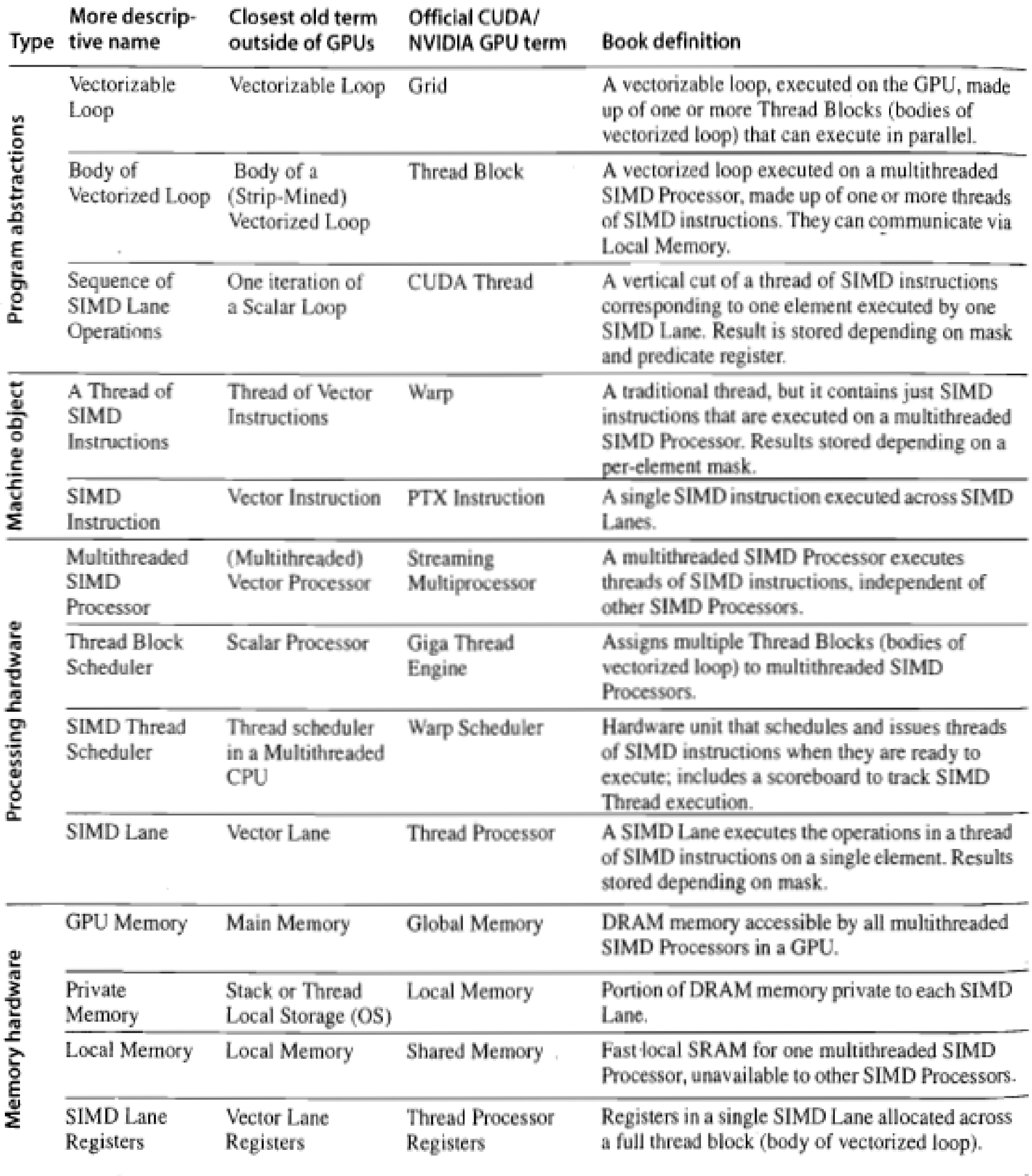
- Basic idea:
  - Heterogeneous execution model
    - CPU is the *host*, GPU is the *device*
  - Develop a C-like programming language for GPU
    - CUDA: Compute Unified Device Architecture
    - C/C++ for host and C/C++ dialect for device (GPU)
  - Unify all forms of GPU parallelism as *CUDA thread*
  - Programming model is “Single Instruction Multiple Thread”

# Threads and Blocks

- A thread is associated with each data element
- Threads are organized into blocks (32 threads)
- Blocks are organized into a grid

- GPU hardware handles thread management, not applications or OS

# Terminology



| Type | More descriptive name | Closest old term outside of GPUs | Official CUDA/ NVIDIA GPU term | Book definition |
|---|---|---|---|---|
| Program abstractions | Vectorizable Loop | Vectorizable Loop | Grid | A vectorizable loop, executed on the GPU, made up of one or more Thread Blocks (bodies of vectorized loop) that can execute in parallel. |
| | Body of Vectorized Loop | Body of a (Strip-Mined) Vectorized Loop | Thread Block | A vectorized loop executed on a multithreaded SIMD Processor, made up of one or more threads of SIMD instructions. They can communicate via Local Memory. |
| | Sequence of SIMD Lane Operations | One iteration of a Scalar Loop | CUDA Thread | A vertical cut of a thread of SIMD instructions corresponding to one element executed by one SIMD Lane. Result is stored depending on mask and predicate register. |
| Machine object | A Thread of SIMD Instructions | Thread of Vector Instructions | Warp | A traditional thread, but it contains just SIMD instructions that are executed on a multithreaded SIMD Processor. Results stored depending on a per-element mask. |
| | SIMD Instruction | Vector Instruction | PTX Instruction | A single SIMD instruction executed across SIMD Lanes. |
| Processing hardware | Multithreaded SIMD Processor | (Multithreaded) Vector Processor | Streaming Multiprocessor | A multithreaded SIMD Processor executes threads of SIMD instructions, independent of other SIMD Processors. |
| | Thread Block Scheduler | Scalar Processor | Giga Thread Engine | Assigns multiple Thread Blocks (bodies of vectorized loop) to multithreaded SIMD Processors. |
| | SIMD Thread Scheduler | Thread scheduler in a Multithreaded CPU | Warp Scheduler | Hardware unit that schedules and issues threads of SIMD instructions when they are ready to execute; includes a scoreboard to track SIMD Thread execution. |
| | SIMD Lane | Vector Lane | Thread Processor | A SIMD Lane executes the operations in a thread of SIMD instructions on a single element. Results stored depending on mask. |
| Memory hardware | GPU Memory | Main Memory | Global Memory | DRAM memory accessible by all multithreaded SIMD Processors in a GPU. |
| | Private Memory | Stack or Thread Local Storage (OS) | Local Memory | Portion of DRAM memory private to each SIMD Lane. |
| | Local Memory | Local Memory | Shared Memory | Fast local SRAM for one multithreaded SIMD Processor, unavailable to other SIMD Processors. |
| | SIMD Lane Registers | Vector Lane Registers | Thread Processor Registers | Registers in a single SIMD Lane allocated across a full thread block (body of vectorized loop). |

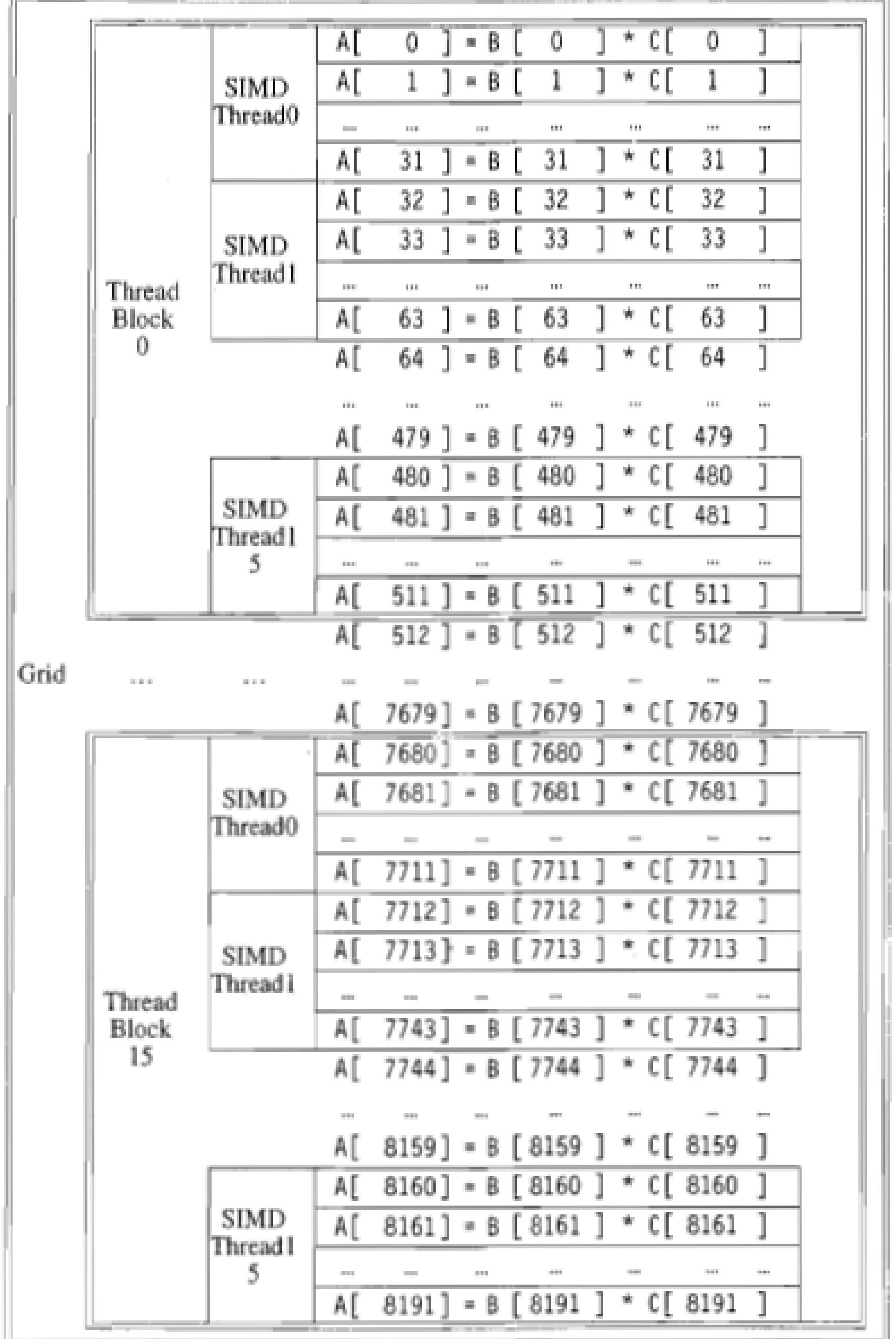
# NVIDIA GPU vs Vector Architectures

- Similarities to vector machines:
  - Works well with data-level parallel problems
  - Scatter-gather transfers
  - Mask registers
  - Large register files
- Differences:
  - No scalar processor
  - Uses multithreading to hide memory latency
  - Has many functional units, as opposed to a few deeply pipelined units like a vector processor

# Exmpl p291

- Multiply two vectors of length 8192 (8K)
  - Code that works over all elements is the grid
  - Thread blocks break this down into manageable sizes
    - Up to 512 elements per block
  - SIMD instruction executes 32 elements at a time (one thread)
    - 8192 elements / 32 (elem/thread) = 256 threads
    - 256 threads = 16 blocks with 16 threads each
  - Thus grid size = 16 blocks  (16 = 8192 / 512)
  - Block is analogous to a strip-mined vector loop with vector length of 32
  - Block is assigned to a *multithreaded SIMD processor* by the *thread block scheduler*
  - Current-generation GPUs (Fermi) have 7-15 multithreaded SIMD processors

# Exmpl p291



Grid

Thread Block 0

SIMD Thread0

A[ 0 ] = B [ 0 ] * C[ 0 ]
A[ 1 ] = B [ 1 ] * C[ 1 ]
... ... ... ... ... ... ...
A[ 31 ] = B [ 31 ] * C[ 31 ]

SIMD Thread1

A[ 32 ] = B [ 32 ] * C[ 32 ]
A[ 33 ] = B [ 33 ] * C[ 33 ]
... ... ... ... ... ... ...
A[ 63 ] = B [ 63 ] * C[ 63 ]

A[ 64 ] = B [ 64 ] * C[ 64 ]
... ... ... ... ... ... ...
A[ 479 ] = B [ 479 ] * C[ 479 ]

SIMD Thread1 5

A[ 480 ] = B [ 480 ] * C[ 480 ]
A[ 481 ] = B [ 481 ] * C[ 481 ]
... ... ... ... ... ... ...
A[ 511 ] = B [ 511 ] * C[ 511 ]

A[ 512 ] = B [ 512 ] * C[ 512 ]

... ... ... ... ... ... ... ... ...

A[ 7679] = B [ 7679 ] * C[ 7679 ]

Thread Block 15

SIMD Thread0

A[ 7680] = B [ 7680 ] * C[ 7680 ]
A[ 7681] = B [ 7681 ] * C[ 7681 ]
... ... ... ... ... ... ...
A[ 7711] = B [ 7711 ] * C[ 7711 ]

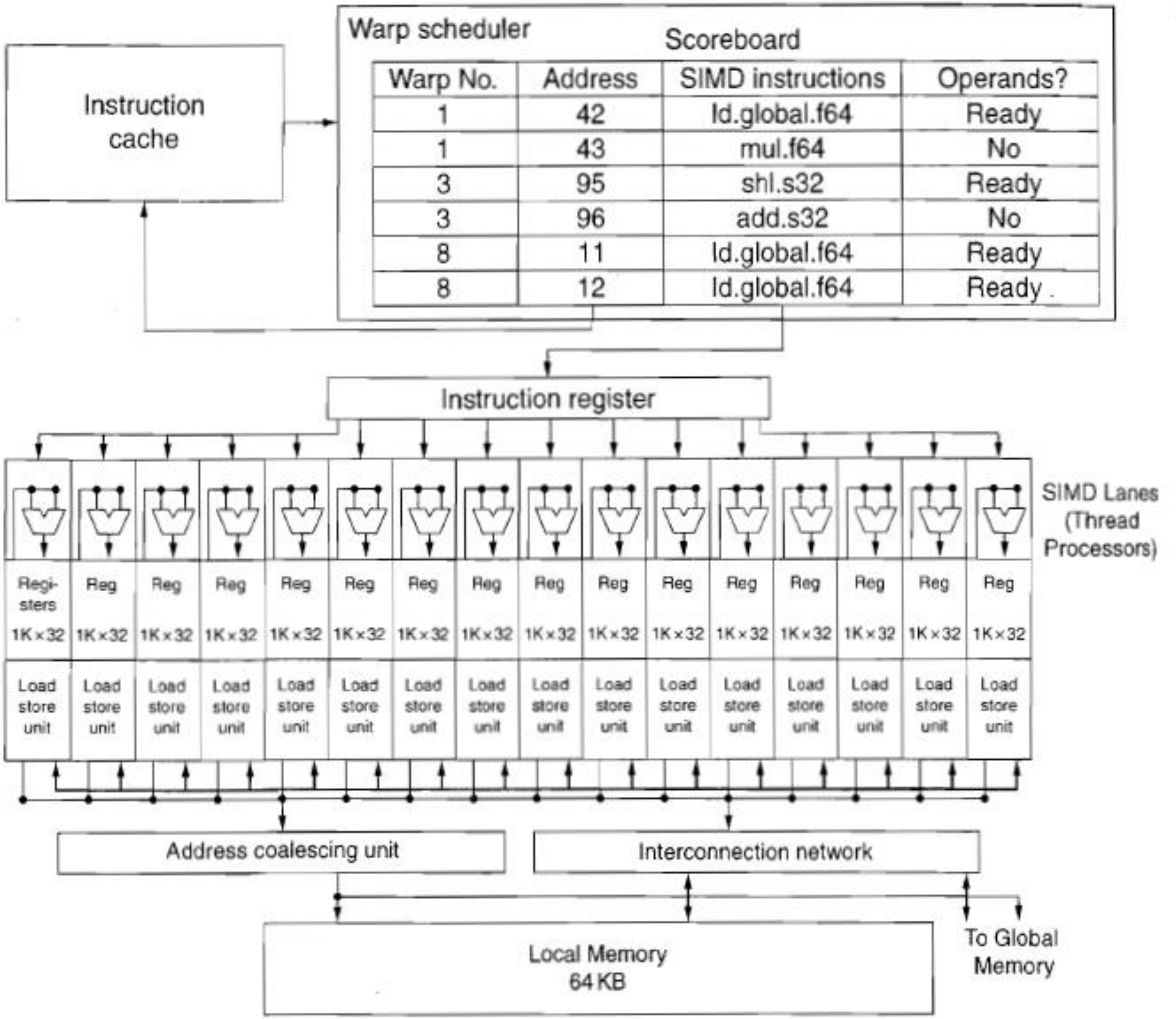
SIMD Thread1

A[ 7712] = B [ 7712 ] * C[ 7712 ]
A[ 7713} = B [ 7713 ] * C[ 7713 ]
... ... ... ... ... ... ...
A[ 7743] = B [ 7743 ] * C[ 7743 ]

A[ 7744] = B [ 7744 ] * C[ 7744 ]
... ... ... ... ... ... ...
A[ 8159] = B [ 8159 ] * C[ 8159 ]

SIMD Thread1 5

A[ 8160] = B [ 8160 ] * C[ 8160 ]
A[ 8161] = B [ 8161 ] * C[ 8161 ]
... ... ... ... ... ... ...
A[ 8191] = B [ 8191 ] * C[ 8191 ]



| Warp No. | Address | SIMD instructions | Operands? |
|---|---|---|---|
| 1 | 42 | ld.global.f64 | Ready |
| 1 | 43 | mul.f64 | No |
| 3 | 95 | shl.s32 | Ready |
| 3 | 96 | add.s32 | No |
| 8 | 11 | ld.global.f64 | Ready |
| 8 | 12 | ld.global.f64 | Ready |

**Figure 4.14 Simplified block diagram of a Multithreaded SIMD Processor.** It has 16 SIMD lanes. The SIMD Thread Scheduler has, say, 48 independent threads of SIMD instructions that it schedules with a table of 48 PCs.

# Floor plan of the Fermi GTX 480 GPU

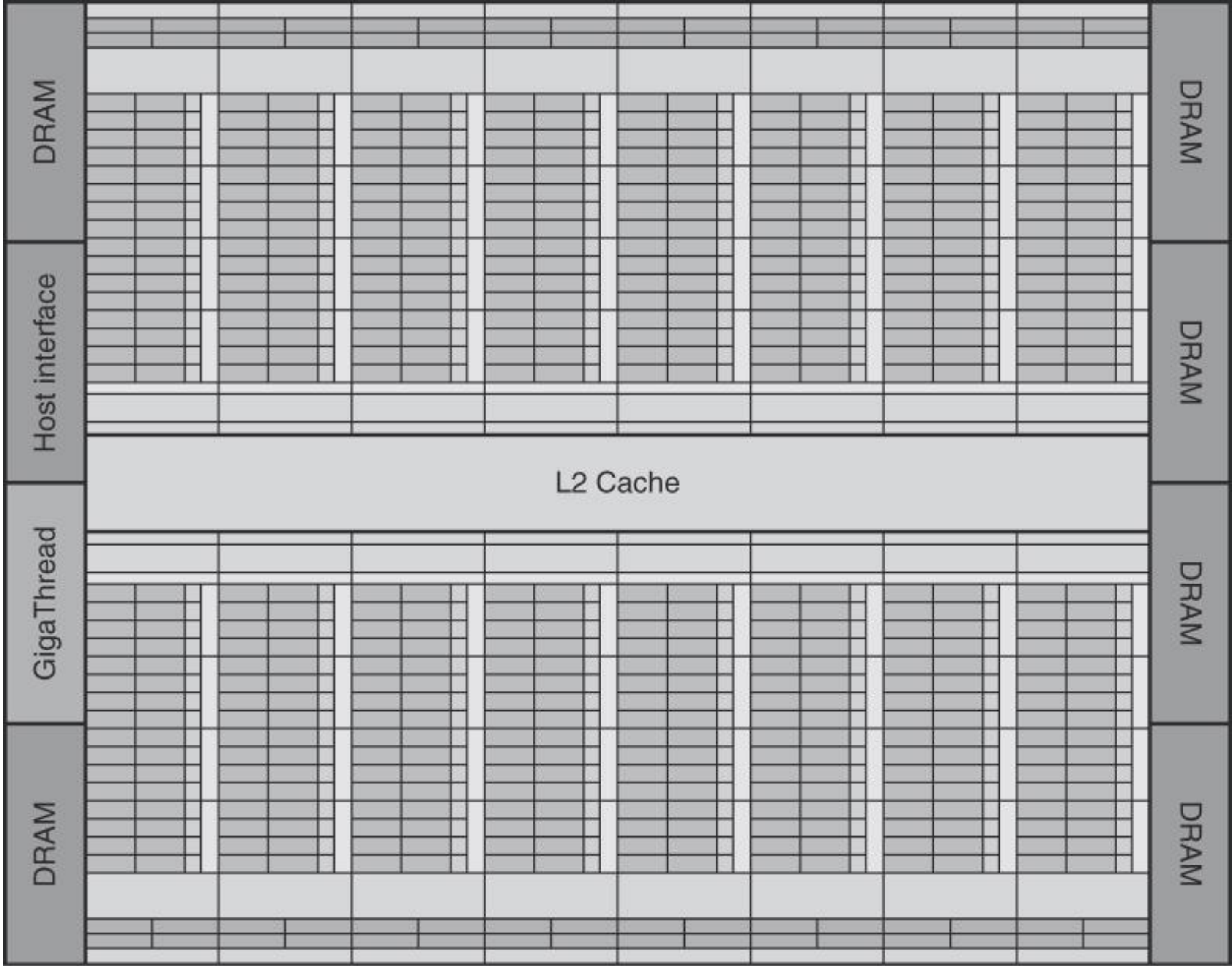


**Figure 4.15.** This diagram shows 16 multithreaded SIMD Processors. The Thread Block Scheduler is highlighted on the left. The GTX 480 has 6 GDDR5 ports, each 64 bits wide, supporting up to 6 GB of capacity. The Host Interface is PCI Express 2.0 x 16. Giga Thread is the name of the scheduler that distributes thread blocks to Multiprocessors, each of which has its own SIMD Thread Scheduler.

# One more level of detail

- *Threads of SIMD instructions*
  - Each has its own PC
  - Thread scheduler uses scoreboard to dispatch
  - No data dependencies between threads!
  - Keeps track of up to 48 threads of SIMD instructions
    - Hides memory latency
- Thread block scheduler schedules blocks to SIMD processors
- Within each SIMD processor:
  - 32 SIMD lanes
  - Wide and shallow compared to vector processors

# Example

- NVIDIA GPU has 32,768 registers
  - Divided into lanes
  - Each SIMD thread is limited to 64 registers
  - SIMD thread has up to:
    - 64 vector registers of 32 32-bit elements
    - 32 vector registers of 32 64-bit elements
  - Fermi has 16 physical SIMD lanes, each containing 2048 registers

# Scheduling of threads of SIMD instructions

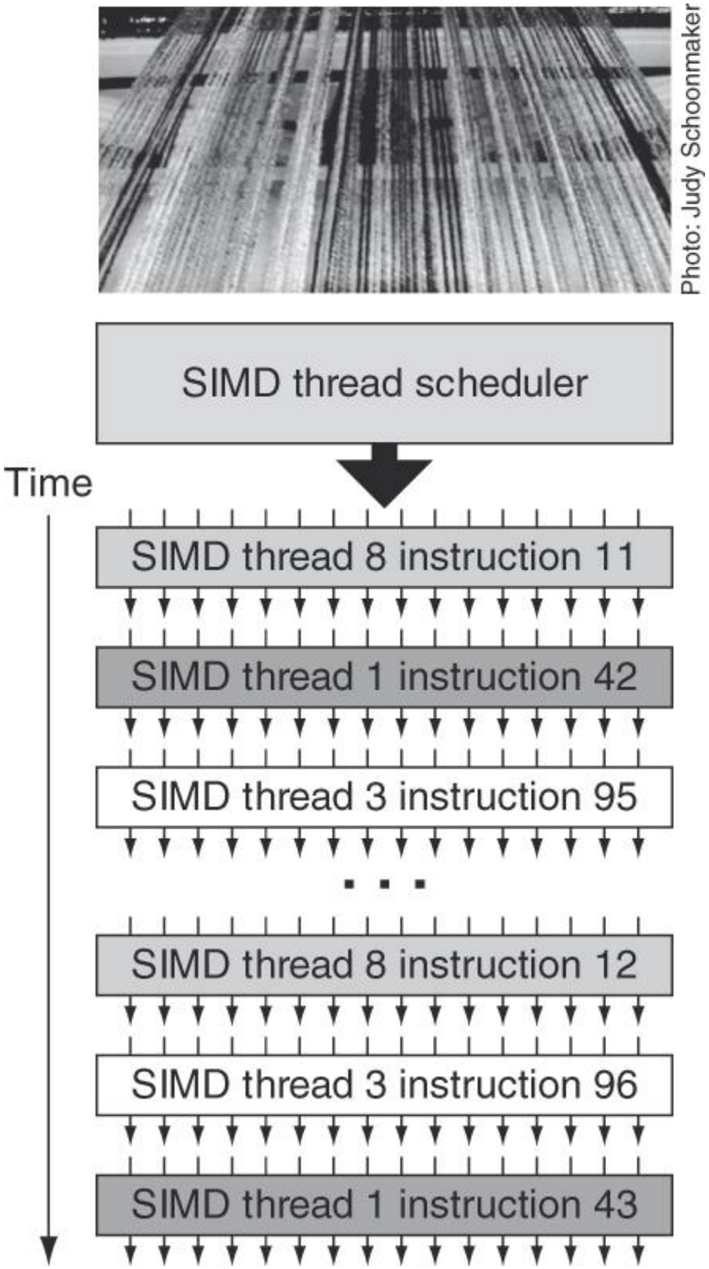


Photo: Judy Schoonmaker
SIMD thread scheduler
Time
SIMD thread 8 instruction 11
SIMD thread 1 instruction 42
SIMD thread 3 instruction 95
SIMD thread 8 instruction 12
SIMD thread 3 instruction 96
SIMD thread 1 instruction 43

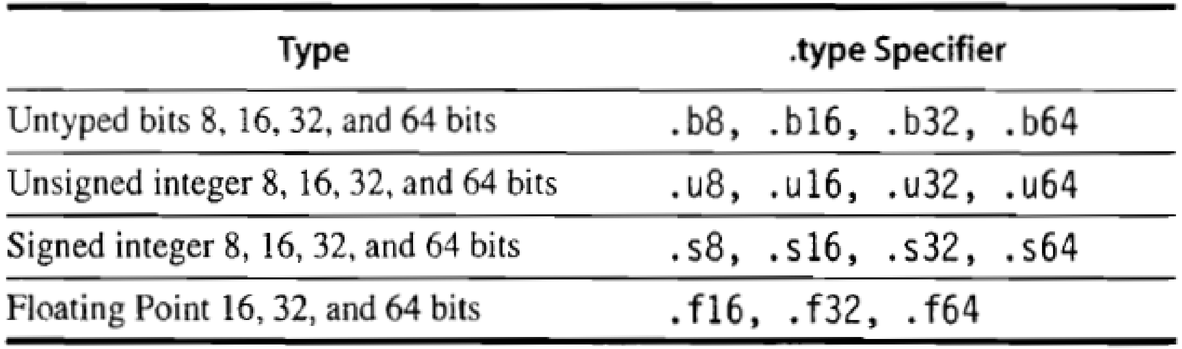
**Figure 4.16.** The scheduler selects a ready thread of SIMD instructions and issues an instruction synchronously to all the SIMD Lanes executing the SIMD thread. Because threads of SIMD instructions are independent, the scheduler may select a different SIMD thread each time.

# NVIDIA Instruction Set Arch.

- PTX  is an abstraction of HW ISA
  - “Parallel Thread Execution (PTX)” stable abstraction in dif. versions
  - PTX → instructions for a single CUDA thread
  - Uses virtual registers; compiler allocates to physical
  - Translation to machine code is performed in software (cf. X86)
- Format

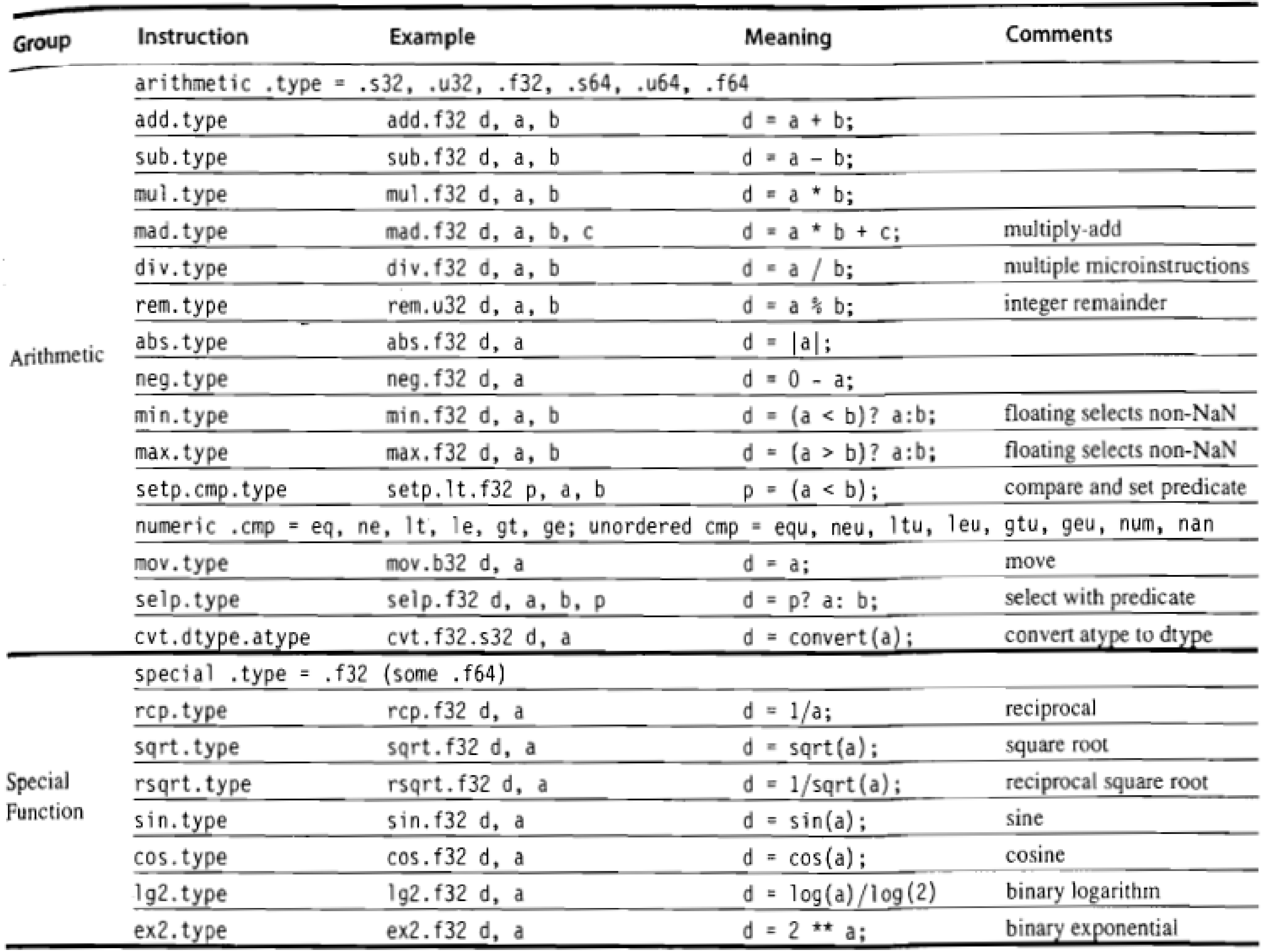
```
opcode.type d, a, b, c;
```

where d = destination operand and a, b, c are source operands

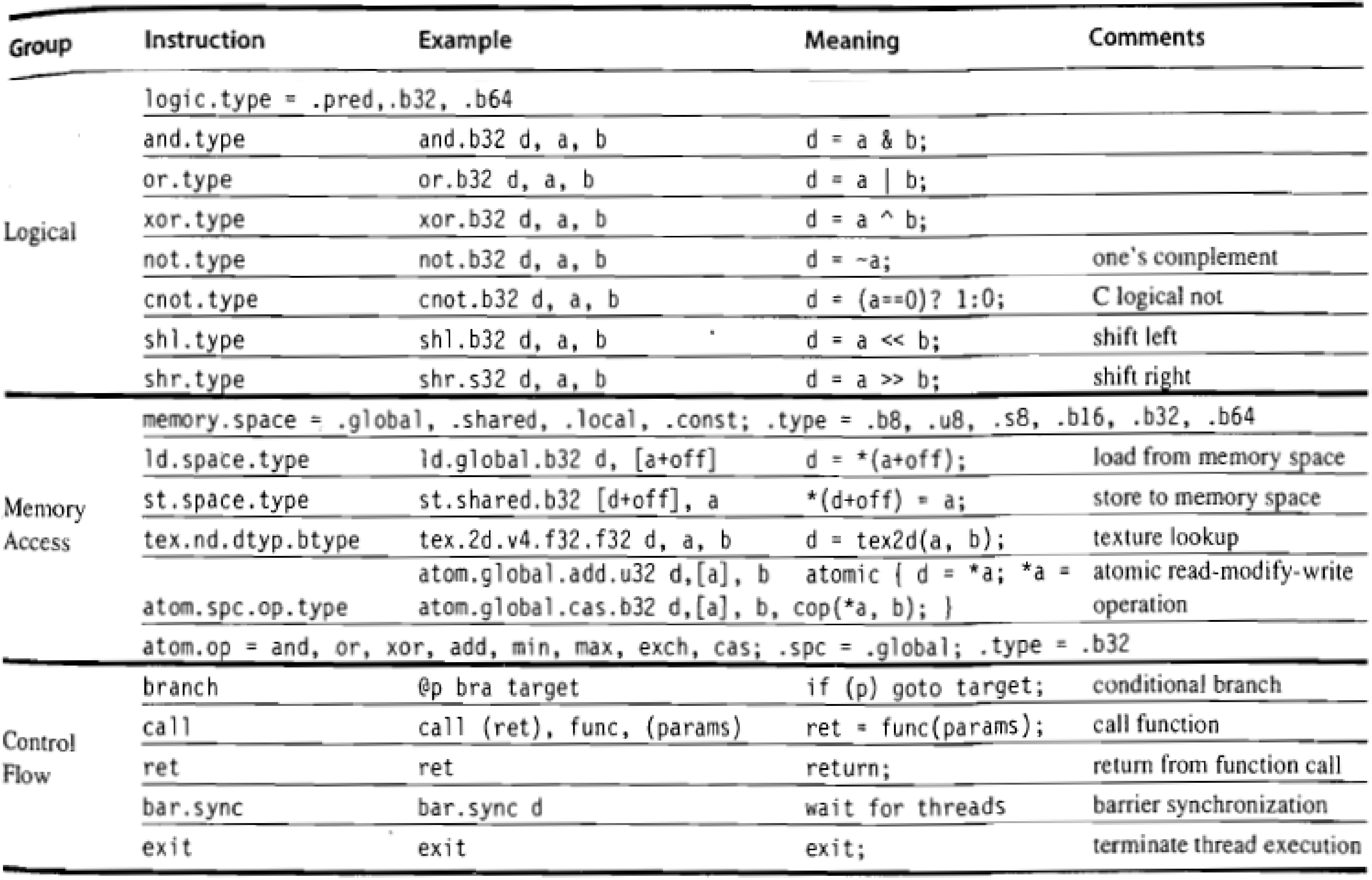


| Type | .type Specifier |
|---|---|
| Untyped bits 8, 16, 32, and 64 bits | .b8, .b16, .b32, .b64 |
| Unsigned integer 8, 16, 32, and 64 bits | .u8, .u16, .u32, .u64 |
| Signed integer 8, 16, 32, and 64 bits | .s8, .s16, .s32, .s64 |
| Floating Point 16, 32, and 64 bits | .f16, .f32, .f64 |

- All instructions: can have 1-bit predicate register
  - equivalent to mask register (see fig 4.21)



| Group | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| Arithmetic | arithmetic .type = .s32, .u32, .f32, .s64, .u64, .f64 | | | |
| | add.type | add.f32 d, a, b | d = a + b; | |
| | sub.type | sub.f32 d, a, b | d = a – b; | |
| | mul.type | mul.f32 d, a, b | d = a * b; | |
| | mad.type | mad.f32 d, a, b, c | d = a * b + c; | multiply-add |
| | div.type | div.f32 d, a, b | d = a / b; | multiple microinstructions |
| | rem.type | rem.u32 d, a, b | d = a % b; | integer remainder |
| | abs.type | abs.f32 d, a | d = \|a\|; | |
| | neg.type | neg.f32 d, a | d = 0 - a; | |
| | min.type | min.f32 d, a, b | d = (a < b)? a:b; | floating selects non-NaN |
| | max.type | max.f32 d, a, b | d = (a > b)? a:b; | floating selects non-NaN |
| | setp.cmp.type | setp.lt.f32 p, a, b | p = (a < b); | compare and set predicate |
| | numeric .cmp = eq, ne, lt, le, gt, ge; unordered cmp = equ, neu, ltu, leu, gtu, geu, num, nan | | | |
| | mov.type | mov.b32 d, a | d = a; | move |
| | selp.type | selp.f32 d, a, b, p | d = p? a: b; | select with predicate |
| | cvt.dtype.atype | cvt.f32.s32 d, a | d = convert(a); | convert atype to dtype |
| Special Function | special .type = .f32 (some .f64) | | | |
| | rcp.type | rcp.f32 d, a | d = 1/a; | reciprocal |
| | sqrt.type | sqrt.f32 d, a | d = sqrt(a); | square root |
| | rsqrt.type | rsqrt.f32 d, a | d = 1/sqrt(a); | reciprocal square root |
| | sin.type | sin.f32 d, a | d = sin(a); | sine |
| | cos.type | cos.f32 d, a | d = cos(a); | cosine |
| | lg2.type | lg2.f32 d, a | d = log(a)/log(2) | binary logarithm |
| | ex2.type | ex2.f32 d, a | d = 2 ** a; | binary exponential |



| Group | Instruction | Example | Meaning | Comments |
|---|---|---|---|---|
| | logic.type = .pred,.b32, .b64 | | | |
| Logical | and.type | and.b32 d, a, b | d = a & b; | |
| | or.type | or.b32 d, a, b | d = a \| b; | |
| | xor.type | xor.b32 d, a, b | d = a ^ b; | |
| | not.type | not.b32 d, a, b | d = ~a; | one's complement |
| | cnot.type | cnot.b32 d, a, b | d = (a==0)? 1:0; | C logical not |
| | shl.type | shl.b32 d, a, b | d = a << b; | shift left |
| | shr.type | shr.s32 d, a, b | d = a >> b; | shift right |
| Memory Access | memory.space = .global, .shared, .local, .const; .type = .b8, .u8, .s8, .b16, .b32, .b64 | | | |
| | ld.space.type | ld.global.b32 d, [a+off] | d = *(a+off); | load from memory space |
| | st.space.type | st.shared.b32 [d+off], a | *(d+off) = a; | store to memory space |
| | tex.nd.dtyp.btype | tex.2d.v4.f32.f32 d, a, b | d = tex2d(a, b); | texture lookup |
| | atom.spc.op.type | atom.global.add.u32 d,[a], b<br>atom.global.cas.b32 d,[a], b, c | atomic { d = *a; *a = op(*a, b); } | atomic read-modify-write operation |
| | atom.op = and, or, xor, add, min, max, exch, cas; .spc = .global; .type = .b32 | | | |
| Control Flow | branch | @p bra target | if (p) goto target; | conditional branch |
| | call | call (ret), func, (params) | ret = func(params); | call function |
| | ret | ret | return; | return from function call |
| | bar.sync | bar.sync d | wait for threads | barrier synchronization |
| | exit | exit | exit; | terminate thread execution |

# Example – DAPY loop

- One iteration

```
shl.s32 R8, blockIdx, 9            ; Thread Block ID * Block size (512 or 2^9)
add.s32 R8, R8, threadIdx          ; R8 = i = my CUDA thread ID
ld.global.f64 RD0, [X+R8]          ; RD0 = X[i]
ld.global.f64 RD2, [Y+R8]          ; RD2 = Y[i]
mul.f64 RD0, RD0, RD4              ; Product in RD0 = RD0 * RD4 (scalar a)
add.f64 RD0, RD0, RD2              ; Sum in RD0 = RD0 + RD2 (Y[i])
st.global.f64 [Y+R8], RD0          ; Y[i] = sum (X[i]*a + Y[i])
```

  - One Thread / loop; one unique id # to each threadblock (blockIdx) and one thread within a block (threadIdx)
  - Creates 8192 CUDA threads; uses unique number to address each element → no incrementing or branching code
  - 3 primeiras instruções: calcula o byte offset em R8 que é somado à base dos arrays
  - GPU não tem instruções especiais para transferência de dados sequencial, por stride e gather-scatter. Tudo é gather-scatter

# Conditional Branching

- Like vector architectures (vector masks by SW), GPU branch hardware uses internal masks (and predicate regs by HW)
- Also uses
  - Branch synchronization stack
    - Entries consist of masks for each SIMD lane
    - I.e. which threads commit their results (all threads execute)
  - Instruction markers to manage when a branch diverges into multiple execution paths
    - Push on divergent branch
  - …and when paths converge
    - Act as barriers
    - Pops stack
- Per-thread-lane 1-bit predicate register, specified by programmer

# NVIDIA GPU Memory Structures

- Each SIMD Lane has private section of off-chip DRAM
  - “Private memory”
  - Contains stack frame, spilling registers, and private variables
- Each multithreaded SIMD processor also has on-chip local memory
  - Shared by SIMD lanes / threads within a block
- Memory shared by SIMD processors is GPU Memory
  - Host can read and write only to GPU memory
- Em vez de usar “working set” in “large caches” GPU usa
  - Small streaming caches
  - extensive multithreading to hide long DRAM latencies
  - computing resources
  - large number or registers
  - vector LD/ST amortize latency across many elements, pay for 1st element and pipeline the rest
- Latest GPUs: small caches as BW filters on GPU memory

# GPU Memory structures

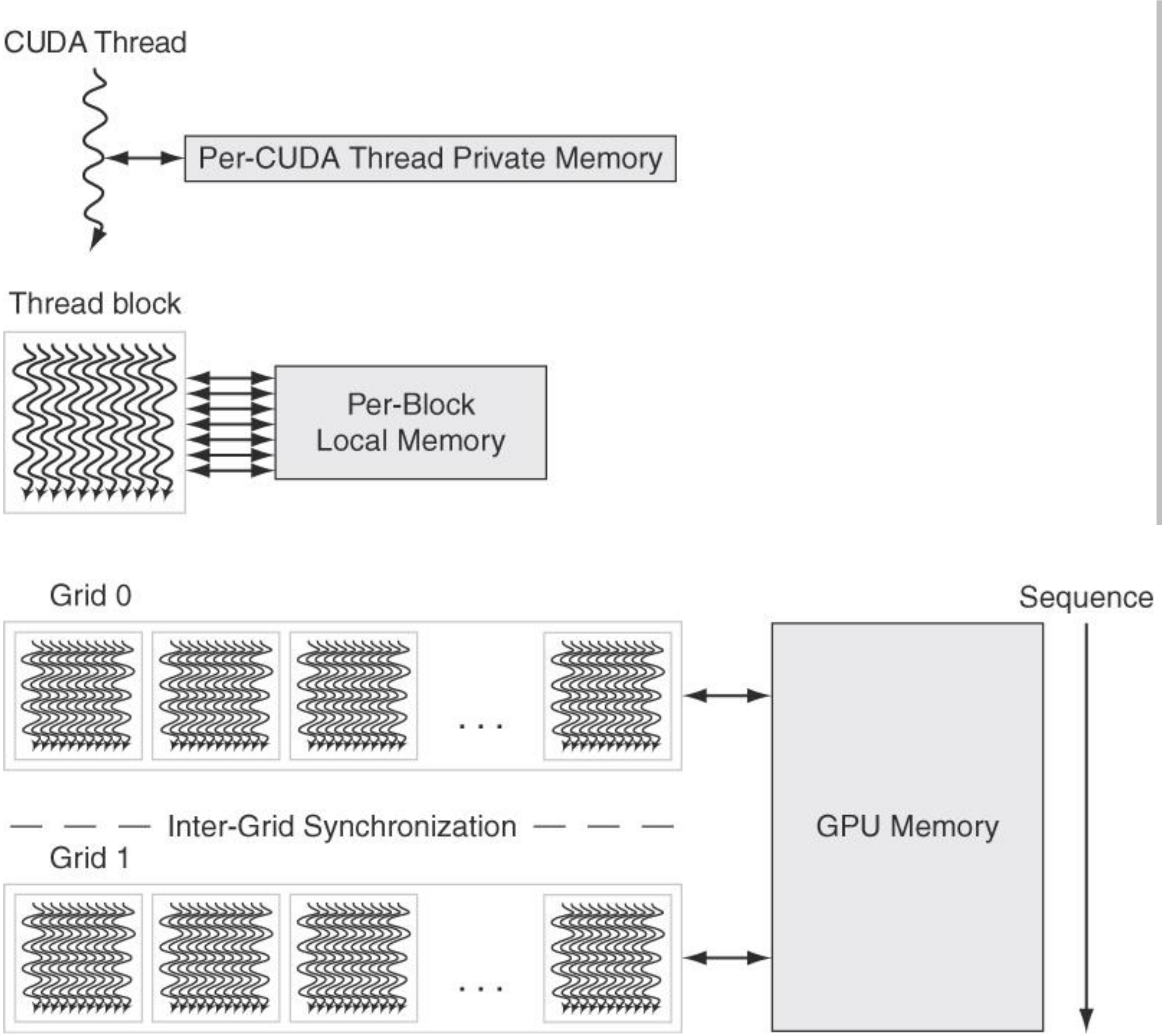
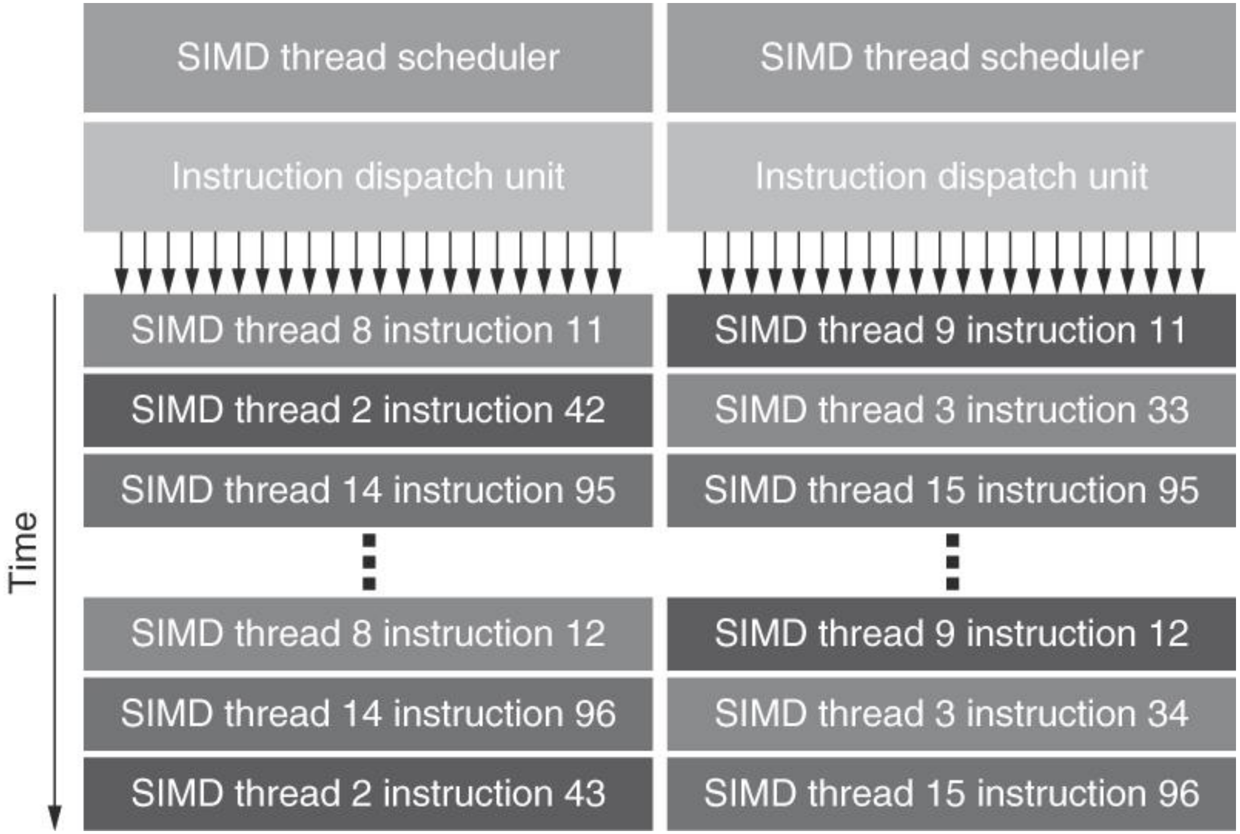


Figure 4.18. GPU Memory is shared by all Grids (vectorized loops), Local Memory is shared by all threads of SIMD instructions within a thread block (body of a vectorized loop), and Private Memory is private to a single CUDA Thread.

# Fermi Architecture Innovations

- Much more complicated than previous versions
- Fermi: each SIMD processor has
  - Two SIMD thread schedulers, two instruction dispatch units (figure)
  - 16 SIMD lanes (SIMD width=32, chime=2 cycles), 16 load-store units, 4 special function units
  - Thus, two threads of SIMD instructions are scheduled every two clock cycles

# Fermi Dual SIMD Thread Scheduler



SIMD thread scheduler | SIMD thread scheduler

Instruction dispatch unit | Instruction dispatch unit

SIMD thread 8 instruction 11 | SIMD thread 9 instruction 11

SIMD thread 2 instruction 42 | SIMD thread 3 instruction 33

SIMD thread 14 instruction 95 | SIMD thread 15 instruction 95

SIMD thread 8 instruction 12 | SIMD thread 9 instruction 12

SIMD thread 14 instruction 96 | SIMD thread 3 instruction 34

SIMD thread 2 instruction 43 | SIMD thread 15 instruction 96

**Figure 4.19** Compare this design to the single SIMD Thread Design in Figure 4.16.

# Fermi Multithreaded SIMD Proc.

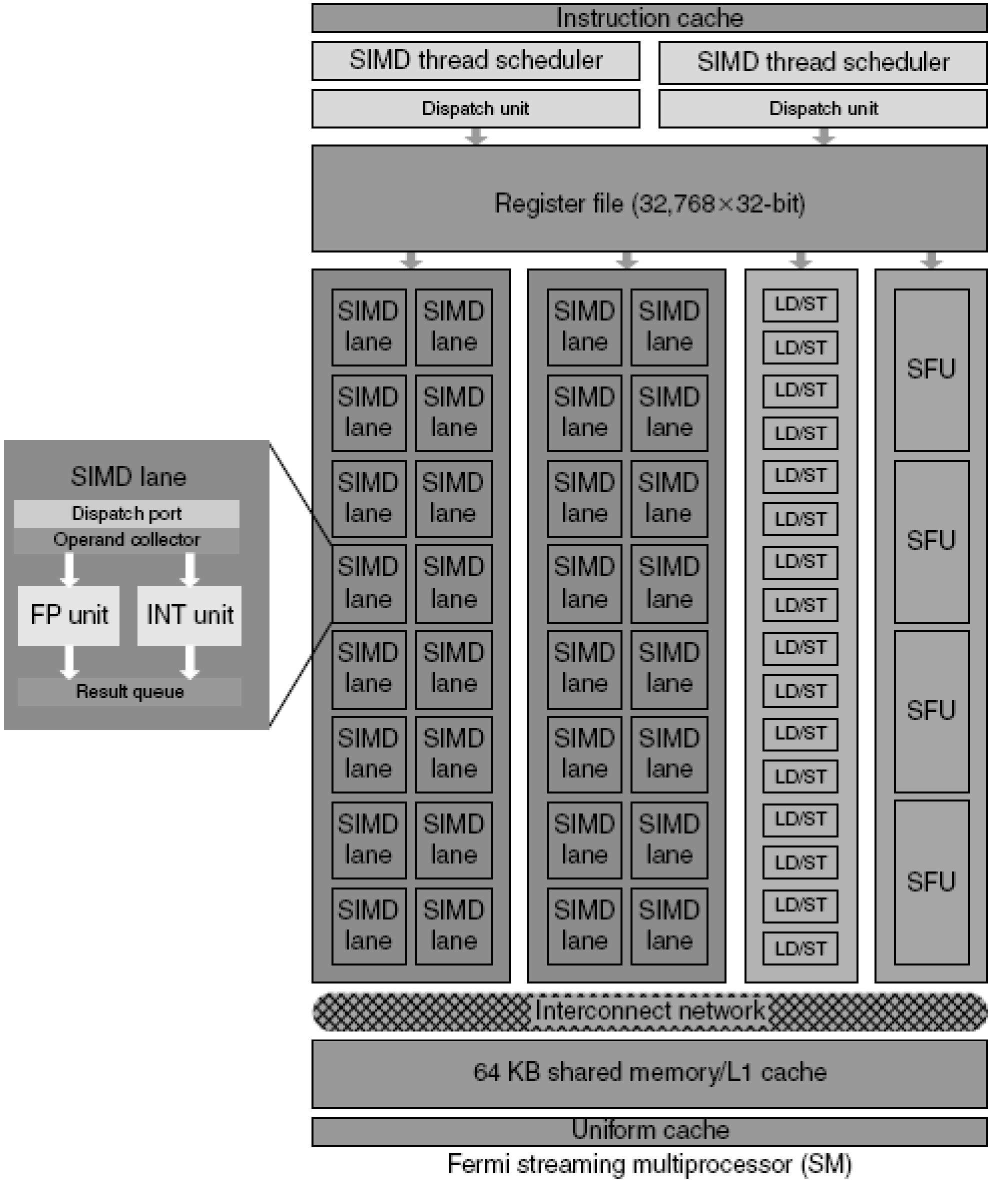


**Figure 4.20 Block diagram of the multithreaded SIMD Processor of a Fermi GPU.** Each SIMD Lane has a pipelined floating-point unit, a pipelined integer unit, some logic for dispatching instructions and operands to these units, and a queue for holding results. The four Special Function units (SFUs) calculate functions such as square roots, reciprocals, sines, and cosines.

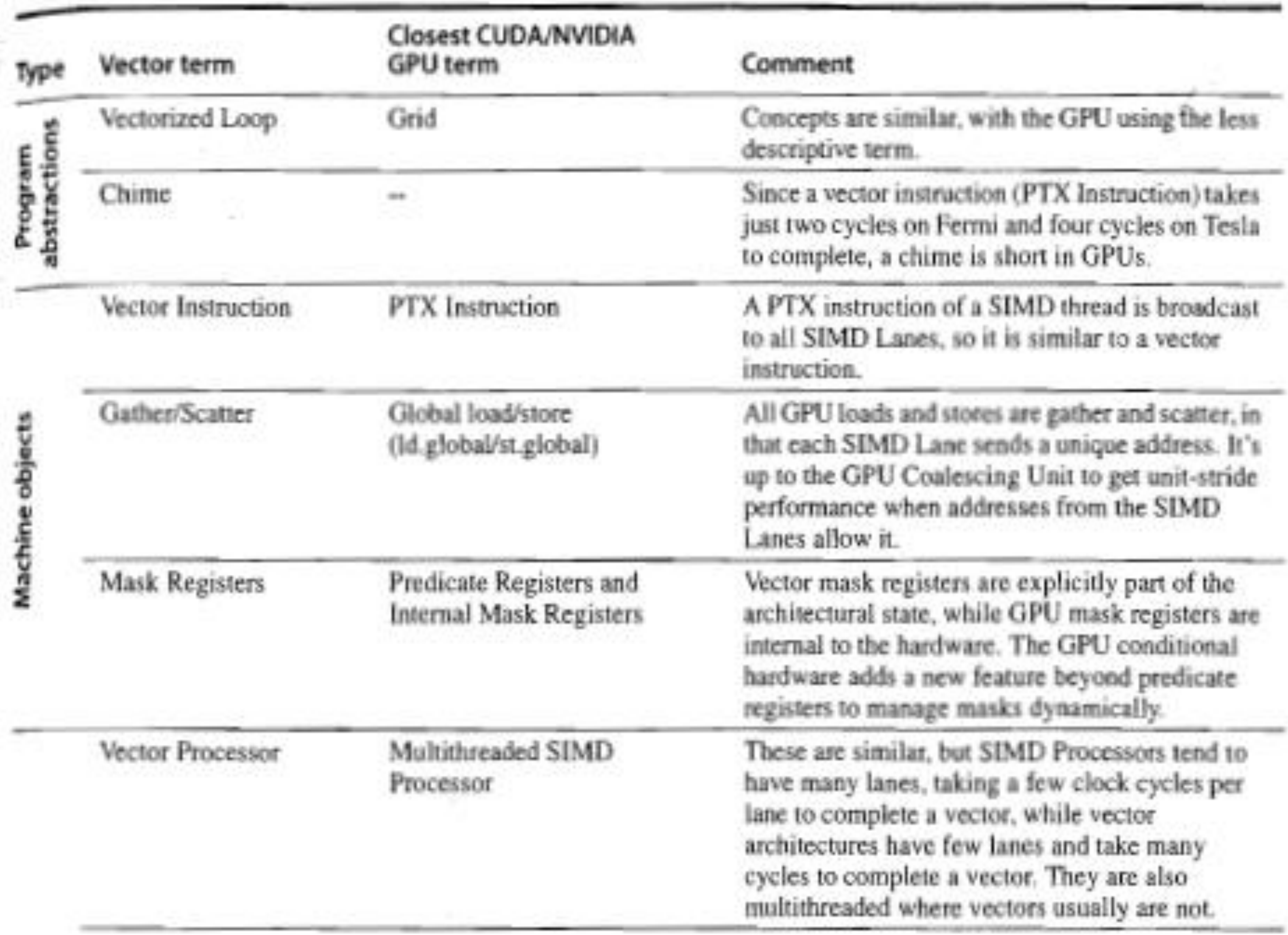
# Fermi Architecture Innovations

- Closer to mainstream system processors
- Fast double precision (DP): rel SP (1/10 prev → ½ now)
  - peak DP performance: 78 GFLOPS (prev) → 515 GFLOPS (now)
- Caches
  - L1 instruction and L1 data caches for each SIMD processor, and a L2 cache shared by SIMD processors and GPU memory. Note: GTX 480 has register file = 2MB and L1 = (0.25 – 0.75 MB)
- 64-bit addressing and unified address space
- Error correcting codes: memory and registers (MTTF?)
- Faster context switching: about 25 μs = 10x faster than in previous versions
- Faster atomic instructions: (5-20)x faster than in previous versions

# Vector Architectures vs GPU

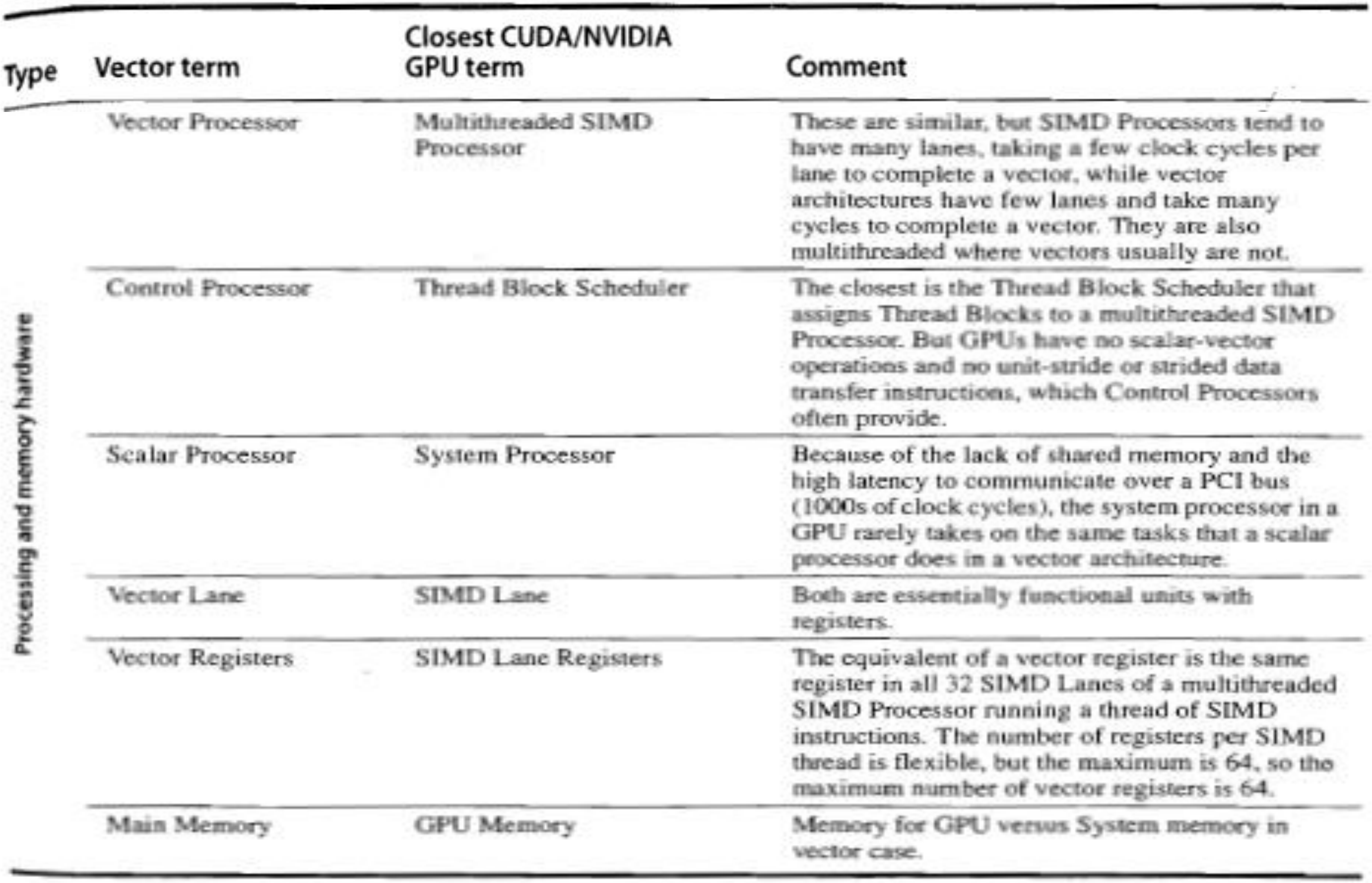
- Many similarities + jargon → confusion: how novel?
- A SIMD Processor is similar to a vector processor
- 1 GPU has multiple SIMD Processors (act as independent MIMD cores)
- NVIDIA GTX 480 is a 15-core machine with hw support for TLP, each core has 16 lanes
- Biggest difference: multithreading (missing for most vector processors)
- Registers:
  - VMIPS: register file holds contiguous entire vectors (8 vectors of 64 elements = 512 elements)
  - GPU: a single vector is distributed across registers of SIMD lanes (1 GPU thread has 64 vectors of 32 elements = 2048 elements → strong support to multithreading)

# Vector Architectures vs GPU: terminology



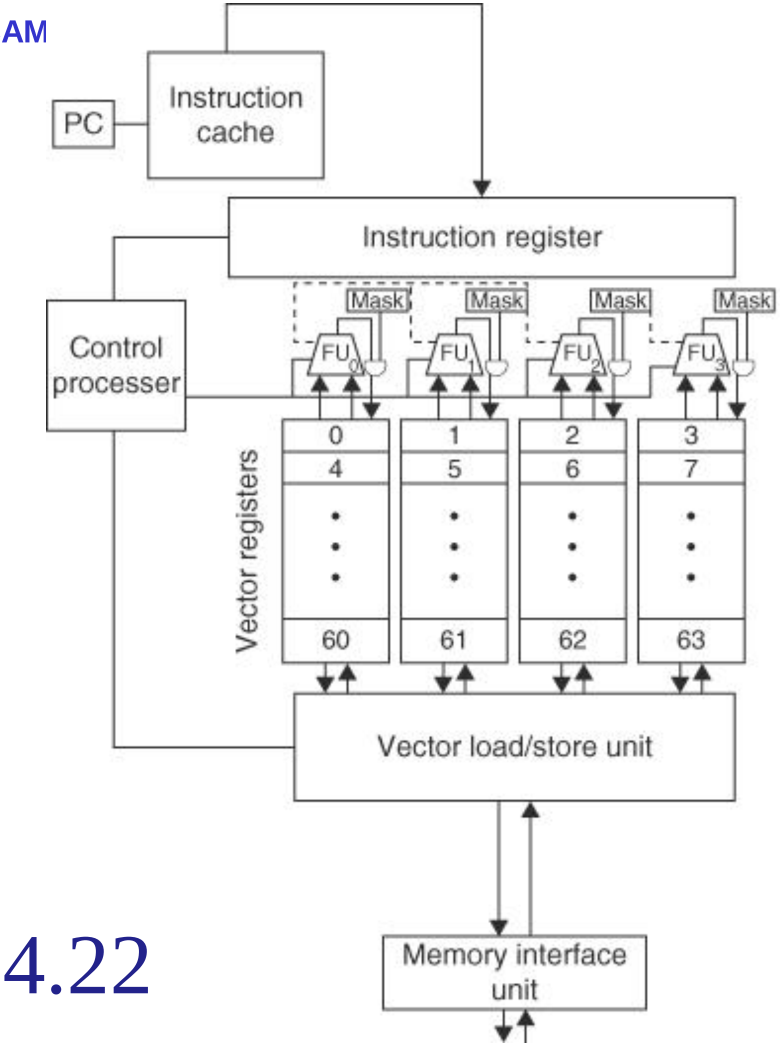
| Type | Vector term | Closest CUDA/NVIDIA GPU term | Comment |
|---|---|---|---|
| Program abstractions | Vectorized Loop | Grid | Concepts are similar, with the GPU using the less descriptive term. |
| | Chime | -- | Since a vector instruction (PTX Instruction) takes just two cycles on Fermi and four cycles on Tesla to complete, a chime is short in GPUs. |
| Machine objects | Vector Instruction | PTX Instruction | A PTX instruction of a SIMD thread is broadcast to all SIMD Lanes, so it is similar to a vector instruction. |
| | Gather/Scatter | Global load/store (ld.global/st.global) | All GPU loads and stores are gather and scatter, in that each SIMD Lane sends a unique address. It's up to the GPU Coalescing Unit to get unit-stride performance when addresses from the SIMD Lanes allow it. |
| | Mask Registers | Predicate Registers and Internal Mask Registers | Vector mask registers are explicitly part of the architectural state, while GPU mask registers are internal to the hardware. The GPU conditional hardware adds a new feature beyond predicate registers to manage masks dynamically. |
| | Vector Processor | Multithreaded SIMD Processor | These are similar, but SIMD Processors tend to have many lanes, taking a few clock cycles per lane to complete a vector, while vector architectures have few lanes and take many cycles to complete a vector. They are also multithreaded where vectors usually are not. |

# Vector Architectures vs GPU: terminology



| Type | Vector term | Closest CUDA/NVIDIA GPU term | Comment |
|---|---|---|---|
| Processing and memory hardware | Vector Processor | Multithreaded SIMD Processor | These are similar, but SIMD Processors tend to have many lanes, taking a few clock cycles per lane to complete a vector, while vector architectures have few lanes and take many cycles to complete a vector. They are also multithreaded where vectors usually are not. |
| | Control Processor | Thread Block Scheduler | The closest is the Thread Block Scheduler that assigns Thread Blocks to a multithreaded SIMD Processor. But GPUs have no scalar-vector operations and no unit-stride or strided data transfer instructions, which Control Processors often provide. |
| | Scalar Processor | System Processor | Because of the lack of shared memory and the high latency to communicate over a PCI bus (1000s of clock cycles), the system processor in a GPU rarely takes on the same tasks that a scalar processor does in a vector architecture. |
| | Vector Lane | SIMD Lane | Both are essentially functional units with registers. |
| | Vector Registers | SIMD Lane Registers | The equivalent of a vector register is the same register in all 32 SIMD Lanes of a multithreaded SIMD Processor running a thread of SIMD instructions. The number of registers per SIMD thread is flexible, but the maximum is 64, so the maximum number of vector registers is 64. |
| | Main Memory | GPU Memory | Memory for GPU versus System memory in vector case. |

## A vector processor with four lanes



## A multithreaded SIMD Processor of a GPU with four SIMD Lanes

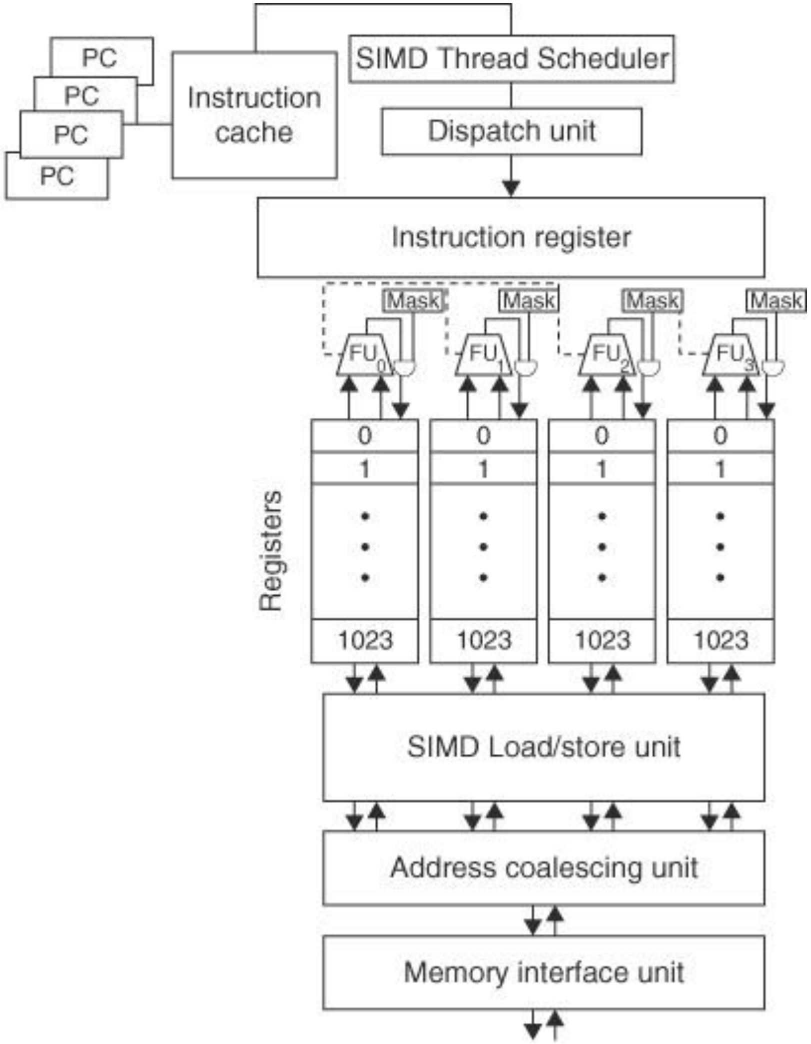


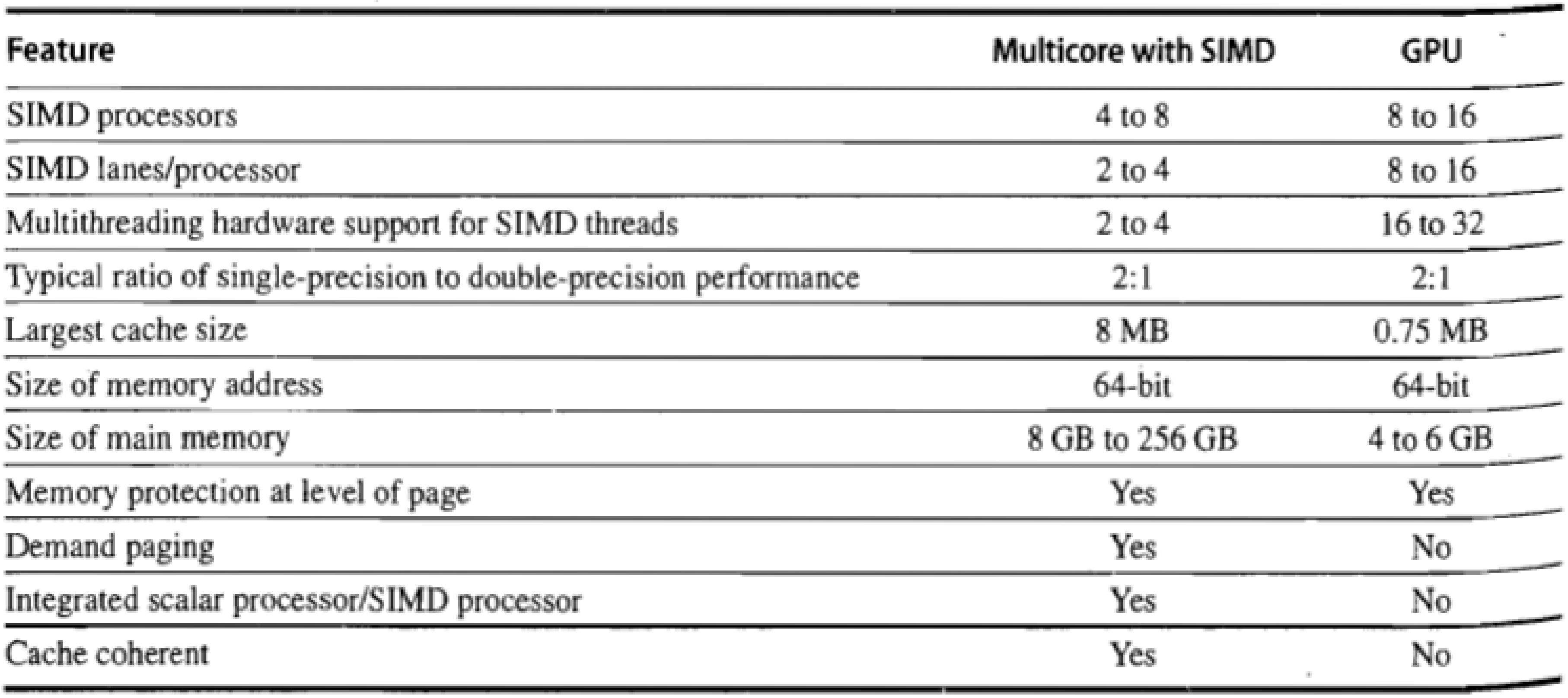
Fig 4.22

(GPUs typically have 8 to 16 SIMD Lanes.) The control processor supplies scalar operands for scalar-vector operations, increments addressing for unit and non-unit stride accesses to memory, and performs other accounting-type operations. Peak memory performance only occurs in a GPU when the Address Coalescing unit can discover localized addressing. Similarly, peak computational performance occurs when all internal mask bits are set identically. Note that the SIMD Processor has one PC per SIMD thread to help with multithreading.

# Multimedia SIMD computers vs GPU

- Both are multiprocessors with multiple SIMD lanes, but GPU has more processors and lanes
- Both use multithreading, but GPU has hw support
- Both use cache, but in GPU they are smaller
- Both use 64-bit address, but GPU has smaller main memory
- Scalar processor:
  - tightly integrated in SIMD multimedia extensions (as in general)
  - separated by I/O bus in GPU
- Support to gather – scatter
  - Multimedia extensions: no
  - GPU: yes

# Multicore multimedia SIMD extension vs GPU



| Feature | Multicore with SIMD | GPU |
|---|---|---|
| SIMD processors | 4 to 8 | 8 to 16 |
| SIMD lanes/processor | 2 to 4 | 8 to 16 |
| Multithreading hardware support for SIMD threads | 2 to 4 | 16 to 32 |
| Typical ratio of single-precision to double-precision performance | 2:1 | 2:1 |
| Largest cache size | 8 MB | 0.75 MB |
| Size of memory address | 64-bit | 64-bit |
| Size of main memory | 8 GB to 256 GB | 4 to 6 GB |
| Memory protection at level of page | Yes | Yes |
| Demand paging | Yes | No |
| Integrated scalar processor/SIMD processor | Yes | No |
| Cache coherent | Yes | No |

**Figure 4.23** Similarities and differences between multicore with Multimedia SIMD extensions and recent GPUs.

# 4.5 Loop-Level Parallelism

- Focuses on determining whether data accesses in later iterations are dependent on data values produced in earlier iterations
  - Loop-carried dependence
- Analyzed (close) at HLL. (ILP usually at Assembly level)
- Here, only RAW is analyzed (naming is easy...)

- Example 1:

```
for (i=999; i>=0; i=i-1)
        x[i] = x[i] + s;
```

- No loop-carried dependence
  - Only within a loop (induction variable): could be eliminated thru loop unrolling

# Exmpl p316: Loop-Level Parallelism

- Ex2: what are data dependences between S1 and S2?

```
for (i=0; i<100; i=i+1) {
        A[i+1] = A[i] + C[i];          /* S1 */
        B[i+1] = B[i] + A[i+1];        /* S2 */
}
```

- Assumes non overlapping arrays
- S1 and S2 use values computed in previous iteration
  - loop carried → successive iterations forced to execute in series
- S2 uses value computed by S1 in same iteration
  - does not prevent different iterations to be executed in parallel
  - could be treated by loop unrolling

# Exmpl p 317: Loop-Level Parallelism

- Ex 3: what are data dependences between S1 and S2?

```
for (i=0; i<100; i=i+1) {
        A[i] = A[i] + B[i]; /* S1 */
        B[i+1] = C[i] + D[i];         /* S2 */
}
```

- S1 uses value computed by S2 in previous iteration (loop carried)
  - but dependence is not circular so loop is parallel
- Loop is parallel if can be written without circular dependence → partial order exists
- No dependence S1→S2; statements can be interchanged
- On 1st iteration, S1 (erro no livro) depends on B[0], calculated prior to initiating the loop

# Exmpl p 317: Loop-Level Parallelism

- Transform to:

```
A[0] = A[0] + B[0];
for (i=0; i<99; i=i+1) {
        B[i+1] = C[i] + D[i];
        A[i+1] = A[i+1] + B[i+1];
}
B[100] = C[99] + D[99];
```

- No more loop carried dependences
  - iterations can be overlapped, provided statements kept in order

# Exmpl p 317: Loop-Level Parallelism

- Ex 4: dependence information could be inexact

```
for (i=0;i<100;i=i+1)  {
        A[i] = B[i] + C[i];
        D[i] = A[i] * E[i];
}
```

- Second reference to A → no need to load, since value already in register
- Aqui é fácil chegar a esta conclusão → ambas referências a A[i] acessariam o mesmo dado na posição de memória → não há intervening access a A[i]
- Em código mais complicado, nem sempre é simples fazer esta análise

# Exmpl p 318: Loop-Level Parallelism

- Example 5: dependência na forma de recorrência

```
for (i=1;i<100;i=i+1)  {
        Y[i] = Y[i-1] + Y[i];
}
```

- Detectar recorrência pode ser importante
  - algumas arquiteturas (vector computers) tem suporte especial para recorrência
  - possível explorar paralelismo ainda no âmbito de ILP

# Finding dependencies

- Dependence analysis complex when: C pointers or Fortran pass by reference.
  - When indices are not affine → x[y[i]] (e.g. sparse matrices)
- Assume indices are affine:
  - $a \text{ x } i + b$ (i is loop index)
- Determining dependence in two references to same array = determining whether two affine functions can have same value for different indices within loop bounds
- Assume:
  - Store to $a \text{ x } i + b$, then
  - Load from $c \text{ x } i + d$
  - $i$ runs from $m$ to $n$
  - Dependence exists if:
    - Given $j$, $k$ such that $m \leq j \leq n$, $m \leq k \leq n$
    - Store to $a \text{ x } j + b$, load from $c \text{ x } k + d$, and $a \text{ x } j + b = c \text{ x } k + d$

# Finding dependencies

- Generally cannot determine at compile time (a, b, c, d unknown)
- Test for absence of a dependence:
  - GCD test:
    - para que uma dependência exista, (*d*-*b*) deve ser divisível por GCD(*c*,*a*)
- Example:

```
for (i=0; i<100; i=i+1) {
    X[2*i+3] = X[2*i] * 5.0;
}
```

- a = 2; b = 3; c = 2; d = 0
  - GCD (a, c) = 2; (d-b) = -3
  - como -3 não é divisível por 2 → não há dependência possível
- O teste de GCD é seguro no resultado negativo mas pode resultar em falso positivo

# Exmpl p 320: Finding dependencies

- Ex 2: find all dependencies, eliminate WAW and WAR by renaming

```
for (i=0; i<100; i=i+1) {
   Y[i] = X[i] / c;      /* S1 */
   X[i] = X[i] + c;      /* S2 */
   Z[i] = Y[i] + c;      /* S3 */
   Y[i] = c - Y[i];      /* S4 */
}
```

- True dependences: S1 → S3, S4 (Y[i]). Not loop carried, but S3 and S4 must wait S1
- Antidependence: S1 → S2 (X[i]), S3 → S4 (Y[i])
- WAW: S1 → S4 (Y[i])

# Finding dependencies (cont)

```
for (i=0; i<100; i=i+1) {
  Y[i] = X[i] / c;  /* S1 */
  X[i] = X[i] + c;  /* S2 */
  Z[i] = Y[i] + c;  /* S3 */
  Y[i] = c - Y[i];  /* S4 */
}
```

- Code with renaming:

```
for (i=0; i<100; i=i+1) {
   T[i] = X[i] / c;    /* Y → T; solve WAW */
   X1[i] = X[i] + c;   /* X → X1; solve WAR */
   Z[i] = T[i] + c;    /* Y → T; solve WAR */
   Y[i] = c - T[i];    /* S4 */
}
```

- After the loop
  - X renamed to X1
  - compiler could fix this

# Dependence Analysis

- Critical for exploiting parallelism
- Loop level parallelism: dependence analysis is the basic tool
- Drawback
  - applies only under a limited set of circumstances, within a loop
- Many situations: very difficult
  - example: referencing arrays with pointer rather than with indices
    - This is one reason why Fortran is still preferred over C and C++ for scientific applications designed for parallel computers
  - example: analyzing references across procedure calls

# Reductions

- Reduction Operation: example dot matrix

```
for (i=9999; i>=0; i=i-1)
        sum = sum + x[i] * y[i];
```

- Not parallel: loop-carried dependence on variable sum
- Can be transformed into 2 loops, one parallel and other partially paral.

```
for (i=9999; i>=0; i=i-1)
    sum [i] = x[i] * y[i];
for (i=9999; i>=0; i=i-1)
    finalsum = finalsum + sum[i];
```

- Second loop = reduction (used also in MapReduce) → hw support in vector computers
- Example: suppose p processors, p ranging from 0 to 9

```
for (i=999; i>=0; i=i-1)
        finalsum[p] = finalsum[p] + sum[i+1000*p];
```

- Note:  assumes associativity! Finally, a simple scalar loop adds the 10 sums

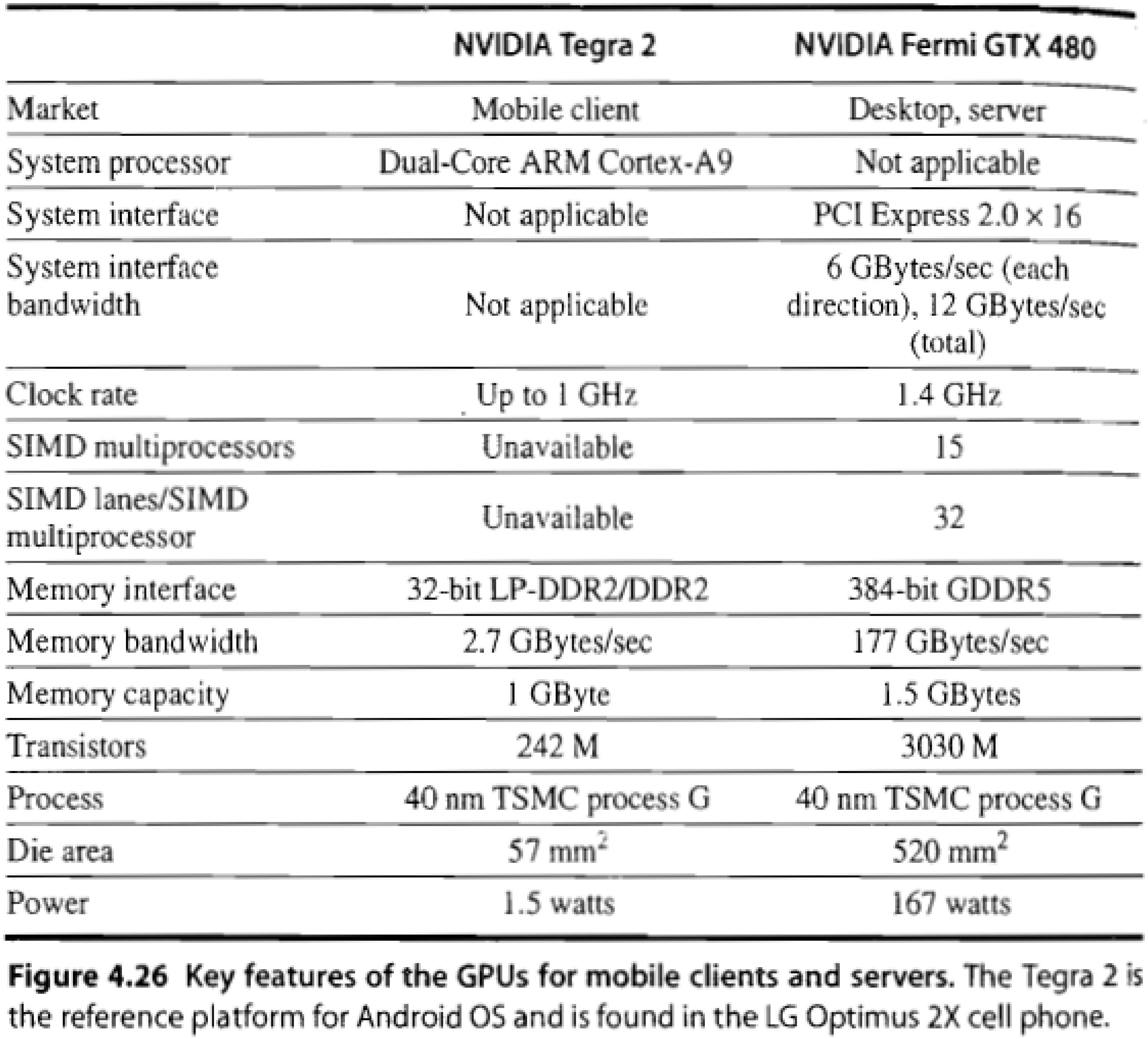
# 4.6 Crosscutting Issues

- Energy and DLP
  - Many FUs, many parallel vector elements, many lanes → high performance with lower clock frequency
  - Compared to out-of-order processors: DLP processors have simpler control logic, no speculation, easier to turn off unused portions of chip
- Banked Memory and Graphics Memory
  - GDRAM: higher bandwidth than conventional DRAM
  - Soldered directly onto GPU board (no DIMM modules)
  - Memory banks → higher bandwidth
- Strided access and TLB misses (VM translations)
  - Problem
  - Depending on TLB organization, array size and striding
    - possible to get one TLB miss for every access to an array element

# 4.7 Putting all together: comparisons

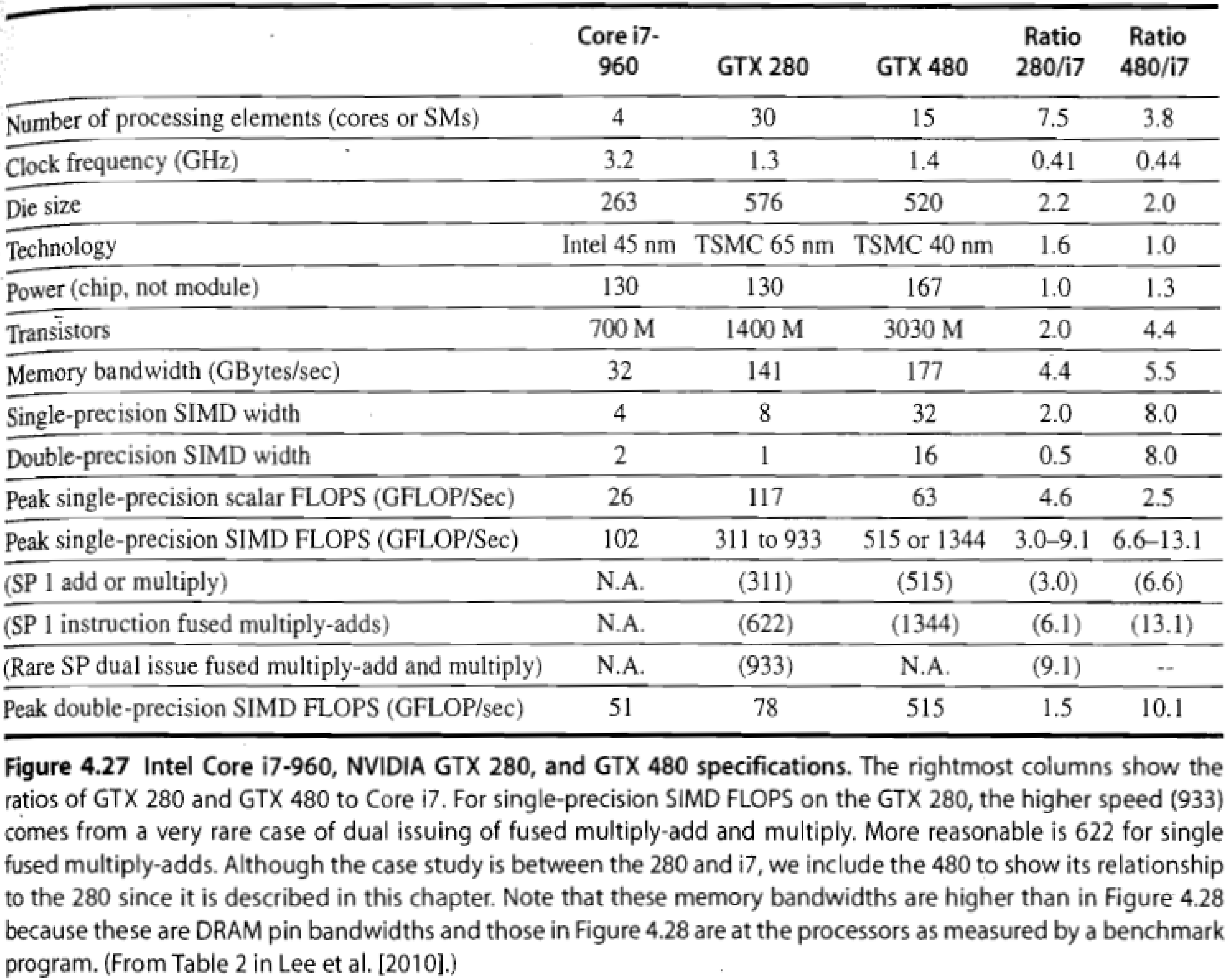
- Mobile versus Server GPUs (Fig. 4.26)
  - Mobile: NVIDIA Tegra 2 → cell phone LG Optimus 2X (Android)
  - Fermi GPU for servers
    - Meta dos engenheiros: animação no servidor 5 anos depois do lançamento do filme; e cinco anos depois no celular
    - Avatar no Servidor GPU em 2015 e no celular em 2020
- Servers: non GPU vs GPU (Fig. 4.27)
  - non GPU: Intel i7 960
  - GPU Server: Fermi GTX 280 and GTX 480

# Fig 4.26: Tegra 2 vs Fermi GTX 480



| | NVIDIA Tegra 2 | NVIDIA Fermi GTX 480 |
|---|---|---|
| Market | Mobile client | Desktop, server |
| System processor | Dual-Core ARM Cortex-A9 | Not applicable |
| System interface | Not applicable | PCI Express 2.0 × 16 |
| System interface bandwidth | Not applicable | 6 GBytes/sec (each direction), 12 GBytes/sec (total) |
| Clock rate | Up to 1 GHz | 1.4 GHz |
| SIMD multiprocessors | Unavailable | 15 |
| SIMD lanes/SIMD multiprocessor | Unavailable | 32 |
| Memory interface | 32-bit LP-DDR2/DDR2 | 384-bit GDDR5 |
| Memory bandwidth | 2.7 GBytes/sec | 177 GBytes/sec |
| Memory capacity | 1 GByte | 1.5 GBytes |
| Transistors | 242 M | 3030 M |
| Process | 40 nm TSMC process G | 40 nm TSMC process G |
| Die area | 57 $mm^2$ | 520 $mm^2$ |
| Power | 1.5 watts | 167 watts |

**Figure 4.26 Key features of the GPUs for mobile clients and servers.** The Tegra 2 is the reference platform for Android OS and is found in the LG Optimus 2X cell phone.

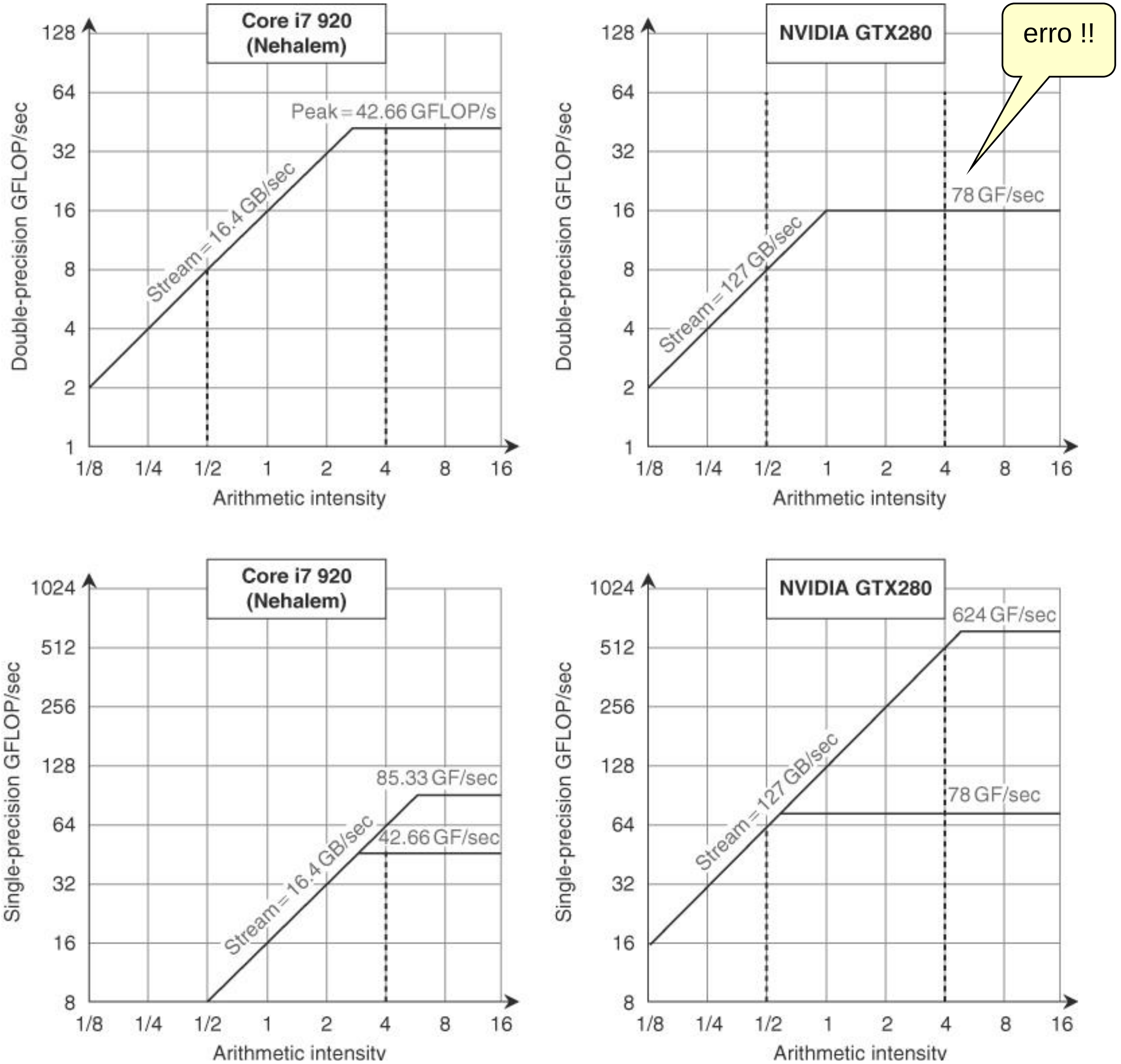


| | Core i7-960 | GTX 280 | GTX 480 | Ratio 280/i7 | Ratio 480/i7 |
|---|---|---|---|---|---|
| Number of processing elements (cores or SMs) | 4 | 30 | 15 | 7.5 | 3.8 |
| Clock frequency (GHz) | 3.2 | 1.3 | 1.4 | 0.41 | 0.44 |
| Die size | 263 | 576 | 520 | 2.2 | 2.0 |
| Technology | Intel 45 nm | TSMC 65 nm | TSMC 40 nm | 1.6 | 1.0 |
| Power (chip, not module) | 130 | 130 | 167 | 1.0 | 1.3 |
| Transistors | 700 M | 1400 M | 3030 M | 2.0 | 4.4 |
| Memory bandwidth (GBytes/sec) | 32 | 141 | 177 | 4.4 | 5.5 |
| Single-precision SIMD width | 4 | 8 | 32 | 2.0 | 8.0 |
| Double-precision SIMD width | 2 | 1 | 16 | 0.5 | 8.0 |
| Peak single-precision scalar FLOPS (GFLOP/Sec) | 26 | 117 | 63 | 4.6 | 2.5 |
| Peak single-precision SIMD FLOPS (GFLOP/Sec) | 102 | 311 to 933 | 515 or 1344 | 3.0–9.1 | 6.6–13.1 |
| (SP 1 add or multiply) | N.A. | (311) | (515) | (3.0) | (6.6) |
| (SP 1 instruction fused multiply-adds) | N.A. | (622) | (1344) | (6.1) | (13.1) |
| (Rare SP dual issue fused multiply-add and multiply) | N.A. | (933) | N.A. | (9.1) | -- |
| Peak double-precision SIMD FLOPS (GFLOP/sec) | 51 | 78 | 515 | 1.5 | 10.1 |

**Figure 4.27 Intel Core i7-960, NVIDIA GTX 280, and GTX 480 specifications.** The rightmost columns show the ratios of GTX 280 and GTX 480 to Core i7. For single-precision SIMD FLOPS on the GTX 280, the higher speed (933) comes from a very rare case of dual issuing of fused multiply-add and multiply. More reasonable is 622 for single fused multiply-adds. Although the case study is between the 280 and i7, we include the 480 to show its relationship to the 280 since it is described in this chapter. Note that these memory bandwidths are higher than in Figure 4.28 because these are DRAM pin bandwidths and those in Figure 4.28 are at the processors as measured by a benchmark program. (From Table 2 in Lee et al. [2010].)

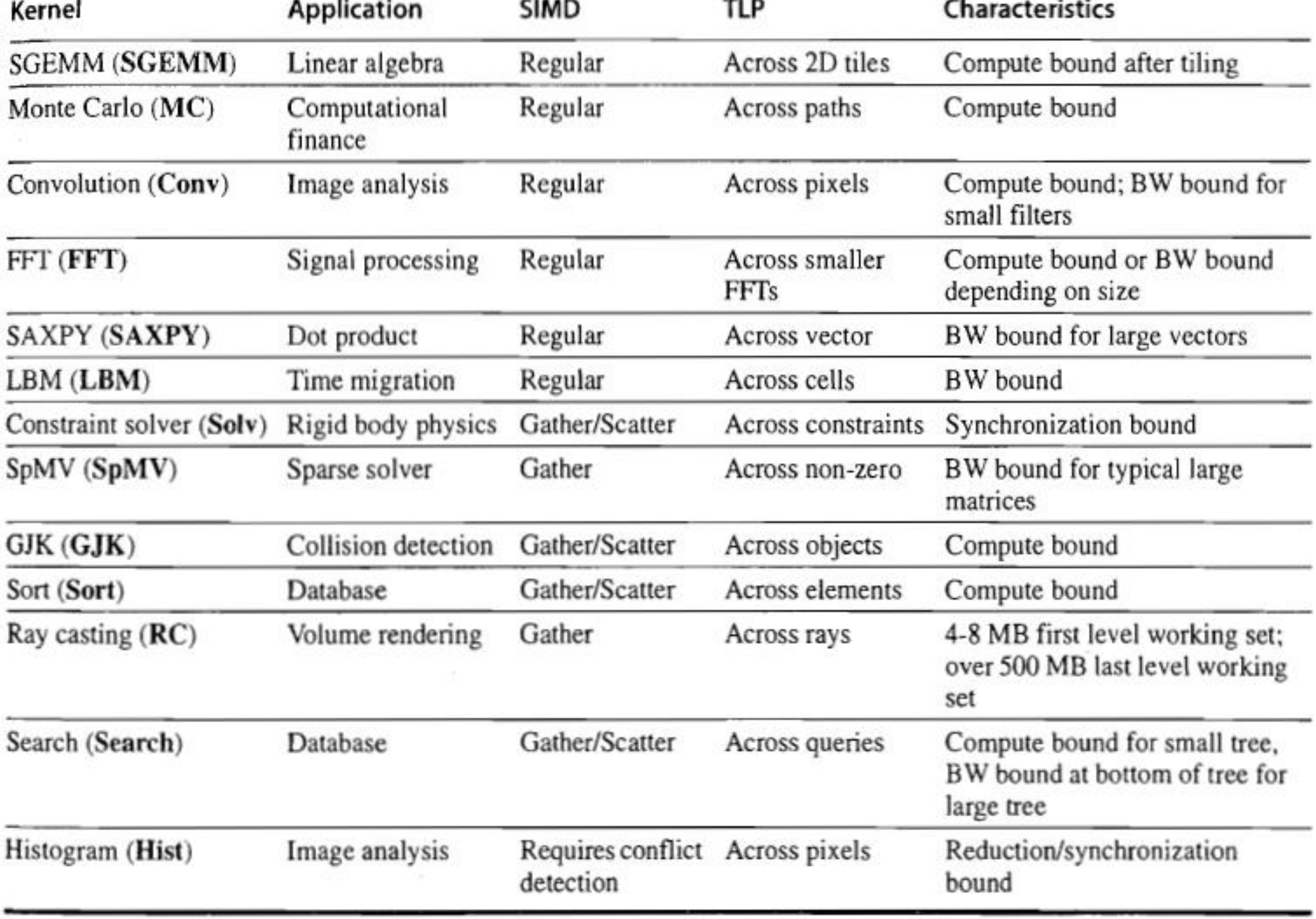
IC-UNICAMP

# Figure 4.28 Roofline model



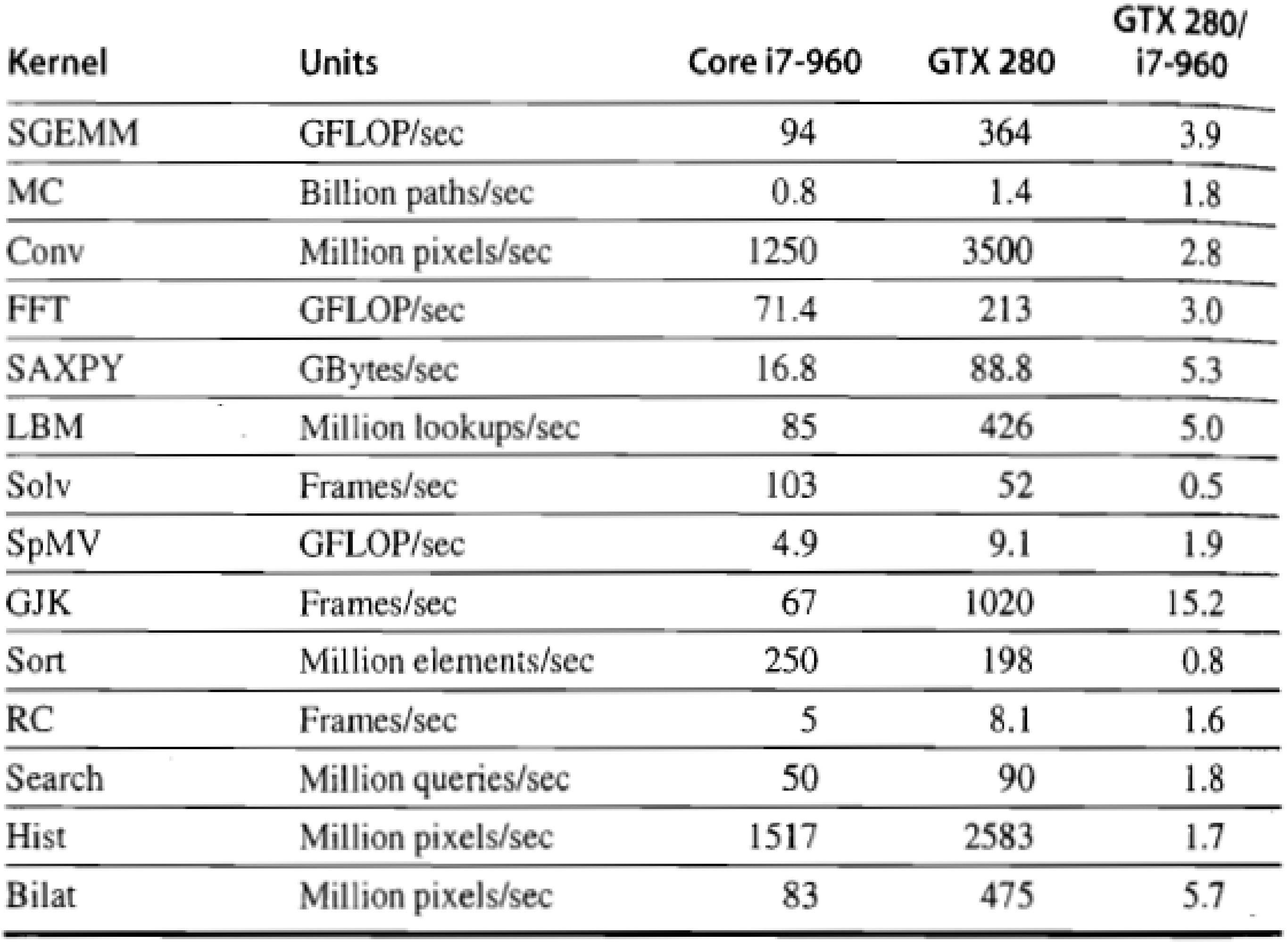
erro !!

These rooflines show double-precision floating-point performance in the top row and single-precision performance in the bottom row. (The DP FP performance ceiling is also in the bottom row to give perspective.) The Core i7 920 on the left has a peak DP FP performance of 42.66 GFLOP/sec, a SP FP peak of 85.33 GFLOP/sec, and a peak memory bandwidth of 16.4 GBytes/sec. The NVIDIA GTX 280 has a DP FP peak of 78 GFLOP/sec, SP FP peak of 624 GFLOP/sec, and 127 GBytes/sec of memory bandwidth. The dashed vertical line on the left represents an arithmetic intensity of 0.5 FLOP/byte. It is limited by memory bandwidth to no more than 8 DP GFLOP/sec or 8 SP GFLOP/sec on the Core i7. The dashed vertical line to the right has an arithmetic intensity of 4 FLOP/byte. It is limited only computationally to 42.66 DP GFLOP/sec and 64 SP GFLOP/sec on the Core i7 and 78 DP GFLOP/sec and 512 DP GFLOP/sec on the GTX 280. To hit the highest computation rate on the Core i7 you need to use all 4 cores and SSE instructions with an equal number of multiplies and adds. For the GTX 280, you need to use fused multiply-add instructions on all multithreaded SIMD processors.



| Kernel | Application | SIMD | TLP | Characteristics |
|---|---|---|---|---|
| SGEMM (**SGEMM**) | Linear algebra | Regular | Across 2D tiles | Compute bound after tiling |
| Monte Carlo (**MC**) | Computational finance | Regular | Across paths | Compute bound |
| Convolution (**Conv**) | Image analysis | Regular | Across pixels | Compute bound; BW bound for small filters |
| FFT (**FFT**) | Signal processing | Regular | Across smaller FFTs | Compute bound or BW bound depending on size |
| SAXPY (**SAXPY**) | Dot product | Regular | Across vector | BW bound for large vectors |
| LBM (**LBM**) | Time migration | Regular | Across cells | BW bound |
| Constraint solver (**Solv**) | Rigid body physics | Gather/Scatter | Across constraints | Synchronization bound |
| SpMV (**SpMV**) | Sparse solver | Gather | Across non-zero | BW bound for typical large matrices |
| GJK (**GJK**) | Collision detection | Gather/Scatter | Across objects | Compute bound |
| Sort (**Sort**) | Database | Gather/Scatter | Across elements | Compute bound |
| Ray casting (**RC**) | Volume rendering | Gather | Across rays | 4-8 MB first level working set; over 500 MB last level working set |
| Search (**Search**) | Database | Gather/Scatter | Across queries | Compute bound for small tree, BW bound at bottom of tree for large tree |
| Histogram (**Hist**) | Image analysis | Requires conflict detection | Across pixels | Reduction/synchronization bound |

**Figure 4.29 Throughput computing kernel characteristics (from Table 1 in Lee et al. [2010].)** The name in parentheses identifies the benchmark name in this section. The authors suggest that code for both machines had equal optimization effort.



| Kernel | Units | Core i7-960 | GTX 280 | GTX 280/<br>i7-960 |
|---|---|---|---|---|
| SGEMM | GFLOP/sec | 94 | 364 | 3.9 |
| MC | Billion paths/sec | 0.8 | 1.4 | 1.8 |
| Conv | Million pixels/sec | 1250 | 3500 | 2.8 |
| FFT | GFLOP/sec | 71.4 | 213 | 3.0 |
| SAXPY | GBytes/sec | 16.8 | 88.8 | 5.3 |
| LBM | Million lookups/sec | 85 | 426 | 5.0 |
| Solv | Frames/sec | 103 | 52 | 0.5 |
| SpMV | GFLOP/sec | 4.9 | 9.1 | 1.9 |
| GJK | Frames/sec | 67 | 1020 | 15.2 |
| Sort | Million elements/sec | 250 | 198 | 0.8 |
| RC | Frames/sec | 5 | 8.1 | 1.6 |
| Search | Million queries/sec | 50 | 90 | 1.8 |
| Hist | Million pixels/sec | 1517 | 2583 | 1.7 |
| Bilat | Million pixels/sec | 83 | 475 | 5.7 |

**Figure 4.30 Raw and relative performance measured for the two platforms.** In this study, SAXPY is just used as a measure of memory bandwidth, so the right unit is GBytes/sec and not GFLOP/sec. (Based on Table 3 in [Lee et al. 2010].)

# Comparação feita pelos engenheiros da Intel

- Memory BW:
  - GPU has 4,4x → LBM (5.0x), SAXPY (5.3x). Working sets too big do not fit into i7 caches. See roofline slopes
- Compute BW
  - 5 benchmarks are compute bound: SGEMM, Conv, FFT, MC, Bilat. 1st three: single precision arith., GPU is 3-6x. MC double precision, GPU only 1.5x. Bilat uses transcendental functions, i7 spends 2/3 of time calculating, GPU 5.7x.
- Cache benefits
  - Ray casting is only 1.6x → cache blocking in i7 prevents it to be memory BW bound
- Gather-Scatter
  - i7 SIMD extension → no benefit if data is scattered. Optimal performance when data is aligned. Biggest difference in GJK = 15.2x
- Synchronization
  - in i7, atomic updates take 28% of total runtime. GTX280 has slow rmw instructions. Synchronization performance can be important for some data parallel problems

# 4.9 Conclusões

- DLP: aumento de importância mesmo em PMD → multimedia
- Previsão:
  - renascimento de DLP na próxima década
  - processadores convencionais (system processors) terão mais características de GPU e vice-versa
- Melhorias esperadas em GPUs
  - Suporte à virtualização
  - Maior capacidade de memória
  - Hoje: I/O → System Memory → GPU Memory. Workloads com muita atividade de I/O se beneficiarão com acesso mais direto
  - Unificação do sistema de memória: alternativa ao bullet anterior